

An-Najah National University
Faculty of Engineering & Information Technology
Department of Computer Engineering



Vision-Assisted Cartesian Robotic System for 3D Block Construction

*A Project Presented to the Faculty of Engineering & Information Technology in Partial
Fulfillment of the Requirements for the Bachelor's degree in Computer Engineering*

Submitted by

Ahmed Taher Gharib
Mohie Aldeen Amjad Halawa
Khalil Mahmoud Qanabita

Supervised by

Dr. Samer Arandi

09/26

Acknowledgements

We would like to thank our supervisor, Dr. Samer Arandi, for the guidance and the patience that carried this project from a sketch of a moving arm to a machine that actually builds something. The advice that mattered most was rarely about code: it was about measuring before believing, and about being honest in writing about what a rig can and cannot do.

We also thank the staff of the Faculty of Engineering & Information Technology for workshop access, for the 3D printing time this project consumed in quantity, and for tolerating a machine that ran forty-second cycles in a shared space for weeks. Finally we thank our families and our classmates, who listened to a great deal of talk about limit switches.

This project was built as a team. The mechanical build, the electronics, the two firmwares, the vision pipeline and the browser software were all shared work, and no part of the system belongs to one of us alone.

Abstract

Small-scale automated assembly is one of those problems that looks solved from a distance and stops looking solved the moment a real machine has to do it. A gantry that moves to a coordinate is straightforward. A machine that takes a structure a person designed, feeds itself one part at a time, picks each part up, turns it the right way round, stacks it on top of the parts already placed, and knows at every moment whether that actually happened, is a different problem, and most of the difference is feedback rather than motion.

This is where our project comes in: a vision-assisted Cartesian robotic cell that builds three-dimensional structures out of 2.2 x 6.0 x 1.5 cm wooden blocks with no human placing a single block by hand. The system is built around three controllers and one brain. A Raspberry Pi 5 is the sole master: it runs the camera, the web server, the orchestration and every safety rule. An Arduino MEGA 2560 runs the gantry, a CoreXY X/Y stage with an added Z axis, a servo claw and a 28BYJ-48 rotation stepper, and executes a fourteen-phase pick-place-park cycle for every block. An Arduino Uno runs a separate feeder module, a two-stage hopper gate, a belt and an alignment servo, and confirms with two HC-SR04 ultrasonic sensors that exactly one block left the hopper and arrived at the fixed pickup point. The two Arduinos never talk to each other; the Pi is the only thing that couples them, and it does so over two independent USB serial links.

The block can be laid either way round, which is why the machine carries two separately calibrated grids and not one: a vertical grid of 7 x 6 addressable cells and a horizontal grid of 3 x 10, sharing one 22.8 x 38.0 cm holder-travel envelope and one feeder cell at [0,0]. An overhead 160-degree fisheye camera watches the build surface through a colour-corrected, lens-corrected pipeline, detects the wooden blocks by warm-colour segmentation, and maps camera pixels to physical centimetres to grid cells through a saved workspace homography.

The operator side is a browser application served from the Pi: a click-to-build console over the live camera image, and a 3D Build Studio in which a person designs a structure, gets immediate physics feedback on support, collisions and toppling, compiles the design into an ordered program of **B col row level** commands separated by grid-mode latches, and runs it while a live digital twin mirrors the real build from the firmware's own phase telemetry.

The completed system placed blocks on hardware with a measured cycle time of 19.8 to 32.2 seconds per block, with the fixed phases of the cycle repeating to within 0.01 s across sixteen logged placements, and stacked a five-level tower on a single cell. The software is covered by 492 browser tests and 68 Python protocol tests. The work demonstrates that the useful engineering in a machine like this is not the kinematics: it is the discipline of never letting one part of the system assume what another part has actually done

Table of Contents

Acknowledgements	1
Abstract	2
Table of Contents	3
1. Introduction	1
1.1 Background and Motivation	1
1.2 Problem Statement	3
1.3 Project Objectives	4
1.4 Scope and Boundaries	5
1.5 Significance of the Project	6
1.6 Organization of the Report	7
2. System Requirements and Design	8
2.1 System Requirements	8
2.1.1 Functional requirements	8
2.1.2 Performance requirements	9
2.1.3 Safety and operational requirements	9
2.2 Design Constraints	10
2.2.1 Physical constraints	10
2.2.2 Hardware and electrical constraints	11
2.3 Overall System Architecture	12
2.4 Mechanical Design	14
2.4.1 Overall structure	14
2.4.2 The X/Y stage: CoreXY	15
2.4.3 The Z axis	17
2.4.4 The end effector	17
2.4.5 The feeder module	19
2.4.6 The workpiece and the build surface	19
2.4.7 The two grids	20
2.4.8 Mechanical problems encountered during construction	22
2.5 Electrical and Control Design	22
2.5.1 Power architecture	22
2.5.2 Wiring and construction	24
2.5.3 Control architecture	24
2.6 System Workflow	25
3. Hardware and System Implementation	27
3.1 Cartesian Motion System	27
3.1.1 Motors and drivers	27
3.1.2 Homing and limit hardware	27
3.1.3 Software limits	28
3.2 End-Effector and Manipulation System	29
3.2.1 The gripper	29
3.2.2 The rotation stepper	29
3.2.3 Tool-centre offsets	30
3.3 Processing and Control Hardware	30
3.3.1 Why a MEGA for the gantry	31
3.4 Vision System	31
3.4.1 Camera hardware and mounting	31

3.4.2 Lens correction	32
3.4.3 Colour correction	33
3.4.4 Calibration approach	34
3.5 Sensors and Feedback	36
3.5.1 Ultrasonic signal acquisition	36
3.5.2 Limit switch acquisition	37
3.6 Power and Safety	37
3.6.1 Power	37
3.6.2 Safety mechanisms that exist	38
3.7 System Integration	38
4. Control and System Operation	41
4.1 Control Methodology	41
4.1.1 A hierarchy, and what closed-loop means here	41
4.1.2 The acknowledgement protocol	41
4.1.3 Real-time build progress	43
4.2 Motion Control	44
4.2.1 Command vocabulary	44
4.2.2 The coordinate system and homing	45
4.2.3 The lattice, and why there are two grids	46
4.2.4 The horizontal registration, and four kinds of offset	47
4.2.5 X-rail skew compensation	47
4.2.6 Z heights and block levels	48
4.3 Vision and Positioning	49
4.3.1 The pipeline, in order	49
4.3.2 Block detection: three layers	50
4.3.3 Grid calibration from the printed target	51
4.3.4 From pixels to a cell	53
4.4 Main System Operation	54
4.4.1 The fourteen-phase build cycle	54
4.4.2 The feed cycle	55
4.4.3 The guard stack	56
4.4.4 The two-board handoff	56
4.5 Additional Operating Modes	58
4.5.1 Browser click-to-build	58
4.5.2 The 3D Build Studio	59
4.5.3 The live digital twin	60
4.5.4 The execution runner	61
4.5.5 Commissioning and simulation modes	62
5. Testing, Results and Discussion	63
5.1 Testing Methodology	63
5.1.1 Four levels, in order	63
5.1.2 Pass and fail criteria	64
5.2 Motion System Testing	64
5.2.1 Cycle repeatability	64
5.2.2 Homing repeatability and lost steps	66
5.3 Manipulation Testing	66
5.3.1 Grip and carry	66
5.3.2 Rotation	66

5.4 Vision System Testing	67
5.4.1 Block detection	67
5.4.2 Calibration results	69
5.4.3 Colour correction	70
5.5 Sensor Testing	71
5.5.1 Ultrasonic sensors	71
5.5.2 Limit switches	71
5.6 Challenges and Limitations	72
5.6.1 The problems that changed the design	72
5.6.2 Limitations of the delivered system	73
6. Conclusion and Future Work	75
6.1 Conclusion	75
6.2 Future Work	76
6.2.1 Safety, first	76
6.2.2 Closing the loop on placement	77
6.2.3 Measurement and calibration	77
6.2.4 Protocol and software	77
6.2.5 Capability	78
References	79
Appendices	80
Appendix A: Bill of Materials	80
Appendix B: Pin Mappings	81
Appendix C: Serial Protocols	82
C.1 Gantry (MEGA) acknowledgement lines	82
C.2 A complete build transaction	83
C.3 Feeder (Uno) protocol 2	83
Appendix D: Firmware Constants	83
Appendix E: Grid Geometry	84
Appendix F: Experimental Data	85

1. Introduction

1.1 Background and Motivation

Automated pick-and-place is one of the oldest jobs in industrial robotics and it is still one of the most common: a machine takes a part from a known place, carries it to a computed place, and lets go. The kinematics of that are well understood, and a student can put a two-axis gantry together from 3D-printer parts in a few weeks. What is much less well understood, and much less often built at student scale, is the layer on top of it. A real assembly cell has to solve where the next part comes from, how it is presented to the tool in a known orientation, what happens when it is not there, how the machine knows a part actually landed, and what the machine is allowed to do next when it does not know.

Most hobby and teaching machines skip that layer entirely, and they skip it in the same way: with time. The belt runs for two seconds because two seconds was usually enough. The gripper closes and the code assumes it closed on something. The arm moves to a coordinate and the program carries on as though it arrived. That works on a bench, in front of a person who can see the machine and stop it, and it stops working the moment the machine has to run a sequence of thirty operations where any one of them can fail quietly. A block that never left the hopper, a block that was gripped by its corner, a block that a hand moved while the gantry was somewhere else: none of these produce an error in a timing-based machine. They produce a wrong structure, built confidently.

We chose a block-stacking gantry because it is the smallest honest version of that problem. Wooden blocks are cheap, identical, and forgiving of a millimetre of error, so the project is not about precision machining. At the same time, stacking is genuinely three-dimensional and genuinely sequential: block seven has to be right or block eighteen falls off it, and there is no way to fake that with a well-chosen demo. The result is a system that is playful to look at, a toy in the best sense, and that has to get the unglamorous parts right in order to work at all.

The mechanical starting point for the X/Y stage is the Instructables **Automated Chessboard** by Greg06 (2022), an open, well-documented CoreXY table that drags magnetised chess pieces around a board with an electromagnet on a moving trolley. We started from it for three concrete reasons rather than for convenience. First, its CoreXY arrangement keeps both motors off the moving beam, so the beam stays light and the working envelope stays large for a given frame, which is exactly the trade a student build wants when it is going to add a third axis and a gripper to that beam later. Second, its homing story is honest and cheap: two limit switches, driven into at every start, defining a repeatable origin without encoders. Third, the reference is explicit that a chessboard does

not need 3D-printer accuracy, and neither does a block-stacking rig with 1.6 cm gaps between cells, so the tolerance budget transfers.

What did not transfer is everything above the table. A chessboard is a plane: the reference never lifts a piece, never turns one, never has to know what is on the board except through a fixed matrix of 64 reed switches under it, and never has to supply itself with pieces. Building in three dimensions turned that reference from a finished machine into a base frame: what we kept from it is the CoreXY X/Y skeleton and its limit-switch homing, and the Z axis, the claw, the feeder, the camera and all three controllers were built on top of that.



Figure 1: The completed rig: CoreXY gantry, Z axis and claw, feeder module on the left, and the overhead camera on its wooden support frame.

1.2 Problem Statement

This project targets one specific problem: **building a human-designed three-dimensional block structure, exactly as designed, with no human placing a single block by hand, and with sensor or vision confirmation at every stage instead of open-loop timing.**

The problem appears the moment the machine has to run unattended for more than one operation. Stacking is sequential and unforgiving: each placement is the foundation of the next, so an error is not an isolated bad block, it is a wrong structure from that point upward. A timing-based machine cannot tell the difference between a successful and a failed operation, so it cannot stop; it carries on and turns one silent failure into a pile. As a result, everything downstream of a wrong placement is wasted, and worse, a machine that continues after an unknown outcome is a machine driving a gripper into a stack whose real height it does not know.

The specific ways an open-loop version of this machine fails are worth naming, because they are what the design has to answer:

- **An empty or jammed hopper.** The gate opens, no block comes out, and the belt runs for its allotted time anyway. The claw then closes on nothing and places nothing, and the machine believes a block is at that cell for the rest of the build.
- **Two blocks instead of one.** A hopper gate that opens far enough for one block often opens far enough for two. The second block sits on the belt behind the first and turns up at the pickup point out of sequence.
- **A block that arrives late or crooked.** A belt is not a positioning system. A block that has not finished travelling, or that has arrived turned a few degrees, is picked by its corner or missed entirely.
- **Lost steps.** Open-loop steppers can lose steps against a stall or a fast direction reversal, and nothing in the motor reports it. If the position counter is trusted across a whole build, that error accumulates cell by cell.
- **A human in the workspace.** A person nudges a block, or takes one back, and the machine's model of the board is silently wrong.
- **A controller reset.** Opening a USB port resets an Arduino. The board comes back un-homed, in its compiled default grid mode, with no memory of anything, and a controller that does not notice keeps sending coordinates into a machine that has forgotten what they mean.

The project focuses instead on making every one of those failures visible and terminal rather than silent, and on making the machine refuse to continue when it cannot prove what physically happened. It does not attempt to make the machine decide **what** to build.

1.3 Project Objectives

This work was done to build a complete, self-feeding robotic cell that constructs a human-designed 3D block structure end to end, and to make every stage of that construction confirmed by a sensor or by the camera rather than by a timer.

The main objectives of this project are:

1. **Provide** a Cartesian gantry that reaches every cell of a calibrated build grid and places a block on any addressable cell, at any stack level, within the machine's own travel envelope. Target: all 41 buildable cells of the vertical grid and all 29 of the horizontal grid reachable, at levels 0 to 16.
2. **Enable** both block orientations, so a structure can mix blocks standing and lying, by turning the claw 90 degrees between the pickup and the placement and carrying a second, separately calibrated grid for the rotated footprint.
3. **Deliver** a feeder module that doses exactly one block at a time from a hopper to a fixed pickup point, and that confirms the block both leaving the container and arriving at the pickup point with independent ultrasonic sensors, so that no placement is ever started on an assumption that a block is there.
4. **Support** an overhead vision system that detects the wooden blocks on the build surface and converts camera pixels into physical centimetres and then into grid cells, so the camera can be used both to calibrate the machine's grid and to verify what actually landed. Target: detect every block on a full board and map a cell to within half a block footprint.
5. **Provide** browser-based operation from a phone or tablet on the local network: a live camera view, click-to-build on the image, and an explicit two-tap confirmation before anything moves, so that the machine can be driven without a keyboard next to it.
6. **Deliver** a 3D design environment in which a person builds the target structure, gets immediate feedback on support, collision and toppling, and compiles the design into an ordered, deterministic command program that the rig then executes block by block.
7. **Enforce** a safety model in which exactly one command runs at a time, never queued; in which a failure that leaves the machine's physical state unknown locks the session and requires a person; and in which no software layer is allowed to retry an operation whose physical outcome it cannot prove.
8. **Achieve** a repeatable cycle time per block. Target: under 45 seconds per block, including feeding, for a machine of this size and step rate.

Autonomous planning of **which** block goes where is deliberately not an objective. The human designs the structure; the build itself, once that design is compiled, is autonomous, and the machine is responsible for executing it exactly and for stopping when it cannot.

1.4 Scope and Boundaries

The scope of this project is one complete robotic cell and the software chain that drives it, from a structure drawn in a browser to a block physically on a stack.

The following are inside the project:

- A Cartesian gantry with CoreXY X/Y motion, an added Z axis, a servo-driven mechanical claw, and a stepper that rotates the claw 90 degrees, giving two block orientations.
- A separate feeder module: a two-stage hopper gate, a conveyor belt, an alignment servo and two ultrasonic sensors, presenting one block at a time at a fixed pickup point.
- An overhead fisheye camera with lens correction, software colour correction, block detection, and camera-to-machine grid calibration.
- A browser operator console served from the Pi, with a live camera view and click-to-build.
- A 3D Build Studio: design, physics validation, compilation to a command program, a live digital twin, and a guarded execution runner with a run report.
- Two independent serial protocols, one per Arduino, and the Pi-side orchestration and safety layers that couple them.

The following aspects are considered out of scope:

1. Autonomous target selection or any form of AI planning. There is no chess engine, no solver, and no block-detection-driven building; the operator or a compiled design chooses every target.
2. A hardwired emergency stop, safety relay, contactor or physical interlock. This is a real and acknowledged gap and is discussed in Sections 3.6 and 5.6.
3. Interrupting a motion already in progress. The gantry firmware does not read serial during a build cycle, so software stop is stop-after-current-block by construction.
4. Metrological camera calibration. The lens model is tuned by eye against straight edges and not fitted from a checkerboard, so the image is visually straight but not measurement-grade.
5. WebRTC or hardware-encoded video. The camera is streamed as motion JPEG over the local network.

6. Real user accounts and multi-operator coordination. A single shared secret on a trusted LAN is the whole access model.
7. Closed-loop motor control. Every stepper in the machine is open-loop; there are no encoders, and a mechanical stall is not detected by the motor itself.

The boundary that matters most is the last one, and it is worth stating precisely because the word closed-loop is used throughout this report. **Closed-loop here means sensor- and vision-confirmed sequencing, not servo position control.** The machine never verifies a motor shaft; it verifies that a block left the hopper, that a block reached the pickup point, that a build phase began and ended, and that a block is visible on the surface where one was commanded. Each of those is a fact the machine measures and not assumes, and each of them can refuse permission for the next step.

1.5 Significance of the Project

The contribution of this project is not the gantry. Cartesian gantries are a solved problem, and ours is a modified version of a published open design. What this project demonstrates is the layer that student-scale automation almost always leaves out: an end-to-end chain in which every physical transition is confirmed by something other than a clock, and in which the machine is designed to stop rather than guess.

Three parts of that chain are worth calling out. The **two-board handoff** is the clearest: a placement is one indivisible operation owned by the Pi, and the gantry is only allowed to move after the feeder has returned one exact terminal message proving a block is staged. There is no timeout that means success and no progress message that counts as permission. The **acknowledged phase protocol** is the second: rather than letting the Pi guess what is happening during a forty-second silence, the firmware narrates its own fourteen phases on a separate machine-readable channel, including its own prediction of how long each Z move should take, which turns an opaque wait into something a user interface and a log can describe honestly. The **design-compile-twin-run chain** is the third: a person draws a structure, the compiler turns it into a deterministic ordered program that respects support order and minimises grid-mode changes, and a digital twin mirrors the real build from the machine's own telemetry rather than from an animation timer.

A system like this is useful to anyone building small automated assembly or sorting cells where parts have to be presented, oriented and stacked: laboratory sample handling, kit assembly, educational robotics, and any teaching setting that wants to show students what the difference between a demo and a machine actually is. It is also directly useful as a teaching artefact in its own right, because every unsafe shortcut the design refuses to take is documented next to the reason it was refused.

1.6 Organization of the Report

This report is organized as follows. **Chapter 2** sets out the requirements the system was designed to, the constraints that shaped it, the overall three-controller architecture, the mechanical and electrical design, and the end-to-end workflow. **Chapter 3** covers the hardware implementation subsystem by subsystem: the motion system and its calibration, the end effector, the three controllers and their division of responsibility, the vision hardware, the sensors, power and safety, and how the whole thing is integrated. **Chapter 4** carries the control side and the main weight of the software: the hierarchical control model, motion control and the grid geometry, the vision-to-coordinate pipeline, the main automatic operating sequence, and the additional operating modes. **Chapter 5** presents the testing methodology and the results, including the measured build timings from the rig's own logs and the automated test coverage, and discusses the challenges and the limitations that remain. **Chapter 6** concludes and sets out future work. The appendices carry the pin maps, the bill of materials, the full command protocols and the raw experimental data.

2. System Requirements and Design

2.1 System Requirements

2.1.1 Functional requirements

The system was designed to eight functional requirements. They are written here as capability statements, in the order the machine actually exercises them during a build.

1. **Place a block at any addressable cell, at any stack level.** One command carries the column, the row and the level, and the firmware turns that into safe step targets itself. The delivered machine addresses 41 buildable cells standing and 29 lying, at levels 0 to 16.
2. **Address the surface as two grids, one per block orientation.** A block measures 2.2 x 6.0 cm in plan and can be laid either way round, and which way round changes how many cells fit and where they sit. The machine carries two separately calibrated grids, switched by a latch command that moves nothing.
3. **Turn the block between pickup and placement.** The claw rotates 90 degrees when the active grid calls for it, and returns to neutral before the next pickup, so a tracked angle is never more than one cycle old.
4. **Dose exactly one block at a time to a fixed pickup point.** A hopper gate that opens in two stages releases the bottom block of a queued column on its own, and shuts behind it before the belt starts so a second block cannot follow.
5. **Confirm the block at both ends of the feed path.** One ultrasonic sensor proves a block physically left the container; a second proves it arrived at the pickup point, and is read again after the aligner moves. A feed request arriving while the pickup point is already occupied is refused before anything opens.
6. **Find the blocks on the surface from overhead, and know which cell each one is in.** The camera pipeline detects the wooden blocks, labels every detection with an integer cell, and converts camera pixels to physical centimetres through a saved calibration.
7. **Operate from a browser on the local network.** A live camera view, cell selection by tapping the image, an explicit two-tap confirmation, and a 3D environment in which a person designs a structure, validates it physically and compiles it to an ordered command program.
8. **Never move without proof.** Exactly one terminal result per command; a failure that moved nothing is reported as a different kind of message from one that may have left a block in the claw; and the whole chain runs against simulated boards with no hardware attached, so the software can be developed and rehearsed without the rig.

2.1.2 Performance requirements

The performance targets came from the machine's own step rates and from what a demonstration needs to be watchable. The **cycle time** target was under 45 seconds per block; the delivered machine averages **26.1 s**, with a fastest cycle of 19.8 s and a worst case of 32.2 s. That figure is dominated by travel and by the Z axis, not by the gripper: the two X/Y moves and the four Z moves account for roughly 20 of those 26 seconds, while the entire grip-and-rotate sequence is under 3.

For **capacity**, the requirement was to use as much of the envelope as the block geometry allows and to stack high enough to be worth watching. The machine addresses the full 22.8 x 38.0 cm holder travel as 41 buildable cells standing or 29 lying, and the firmware ceiling is level 16 at 24.0 cm, leaving 2.5 cm of clearance to fly a block over the tallest possible tower.

For **vision**, the requirement was to find every block on a full board and to map a cell to better than half a block footprint. Detection finds 29 of 29 on both reference boards while correctly rejecting the aluminium rails and two wooden offcuts that are the same shape as blocks, at 37 to 103 ms per analysed frame, and the saved camera-to-cell mapping carries 1.25 px mean and 2.07 px maximum error, which is 0.27 cm on a 2.2 cm block.

2.1.3 Safety and operational requirements

The safety model of this machine is a software model, and it is built on the fact that the gantry cannot be interrupted once it is moving. Every rule below follows from that one physical fact rather than from a general principle.

1. **One command at a time, never queued.** The gantry firmware does not read serial inside its build cycle, so a second command sent during a build sits in a 64-byte hardware buffer and executes late, out of context. The Pi holds a non-blocking lock and refuses any overlapping operation instead of buffering it.
2. **A failure of unknown physical outcome locks the session.** An abort, a timeout, a board reset or a cable loss leaves the claw possibly holding a block at an unknown position. The controller sets a lock, refuses every further mutation, and the only recovery is a person inspecting the rig and restarting the service. There is no retry button, because a retry is the thing that breaks the machine.
3. **Nothing is auto-retried across the two boards.** Once the feeder has staged a block, even a perfectly safe gantry rejection locks the cell: a block is already at the pickup point, so feeding another would double-load it.
4. **Cell [0,0] is the feeder and is never a build target.** It is refused in the firmware, in the Pi and in the browser, in all three, because placing a block there would drop it on the stack the claw picks from.

5. **Every move is preceded by a re-home.** Position is trusted only immediately after a physical switch has defined it.
6. **A build is never started from a stale camera frame or an unhomed axis.** Selection and build are both refused server-side, and the browser is never trusted to enforce either.
7. **Two deliberate taps to move.** Selecting a cell moves nothing and shows the exact command; a separate confirmation sends it.
8. **The machine is operated attended.** With no hardwired emergency stop fitted, the documented procedure is that a person stays with the rig during a run, keeps hands clear of the gantry envelope while it is powered, and stops it by removing power at the supply. This is stated plainly rather than dressed up: it is a real limitation and it is discussed in Sections 3.6 and 5.6.

2.2 Design Constraints

2.2.1 Physical constraints

The machine had to sit on an ordinary table and stay there, and that decision came before any other. It sets the frame, and the frame sets everything downstream: how far the gantry can travel, how many cells fit inside that travel, where the feeder can stand, and how high the camera has to be to see the whole surface at once.

The frame is built from a single 3.6 m length of aluminium profile, cut into the pieces the design needed, and the cut list is the clearest statement of the machine's size:

- **4 legs**, 15 cm each, which is what holds the gantry above the build surface.
- **3 members along X**, 30 cm each.
- **2 members along Y**, 60 cm each.
- **2 members for the feeder**, 30 cm each, one on either side of the conveyor.
- **1 member for the Z column**, approximately 30 cm.

The 60 cm members along Y and the 30 cm members along X are the reason the holder travel comes out at 38.0 cm and 22.8 cm: the difference between the raw member length and the usable travel is the width of the carriages, the pulley mounts and the end brackets

The build area is not simply the travel. It would be easy to assume the reachable surface is whatever the belts can cover, and it is not: a cell is only real if the whole block fits on the machine, not just its centre. A cell is a block footprint plus a uniform 1.6 cm gap, giving a pitch of 3.8 cm across the short axis and 7.6 cm along the long one, and the firmware checks the block edges against a per-mode overhang budget before it accepts a grid at all. That check is what fixes the counts in Section 2.4.7: an eighth vertical column, a seventh vertical row, a fourth horizontal column and an eleventh horizontal row are each refused, not because the motors cannot reach them, but because the block sitting at that cell would

hang off the machine. What survives fills the travel exactly on both axes, $6 \times 3.8 = 22.8$ cm and $5 \times 7.6 = 38.0$ cm.

Two other decisions followed from the same table. The feeder stands beside the gantry rather than hanging above the build surface, because the camera already occupies the space above it and a hopper over the work area would drop blocks into the structure being built. And the camera had to see the whole surface from a height that still cleared the gantry, which put it about 50 cm up and made a wide-angle lens unavoidable, with the distortion and colour cost that Chapter 3 then has to answer.

One physical constraint was not designed for and had to be answered afterwards. The arm holder that rides the X rail is not supported symmetrically: its own mass, the drag of the cable run and the side load from the belt all pull on one side of it, and that constant sideways pull leaves the X rail very slightly out of square with Y. The carriage also drifts along Y in proportion to how far it has travelled along X, and that drift measures about 0.1 cm of Y per column. Re-machining the rail was out of scope, so the drift is cancelled in firmware instead; the model and the correction are in Section 4.2.5.

2.2.2 Hardware and electrical constraints

- **8 KB of SRAM on the AVR.** The gantry sketch prints a great deal of text, and on AVR every plain string literal handed to `Serial.print()` is copied into SRAM at boot. Written naively the sketch needed 9,443 bytes of SRAM, which is 115 % of the chip; it would not have run. Wrapping every literal in `F()` moved them into flash and brought SRAM down to 2,099 bytes (26 %), at a cost of about 1.4 KB of program space (40,590 bytes, 15 % of flash). This is why the serial protocol is terse **key=value** text and not JSON.
- **9600 baud.** The link runs at 9600 8N1, which is roughly 20 ms per line of telemetry. That is the direct reason the firmware reports one line per build phase rather than one per motor step: fourteen lines is about 0.3 s of airtime inside a 40-second build, while per-step telemetry would be minutes of it and would starve the terminal acknowledgement that actually matters.
- **Opening the USB port resets the Arduino.** The board comes back un-homed, in its compiled default grid mode, with no memory of anything the controller had set. There is no EEPROM in use, so the mode latch, the grid size and any live grid shift have to be re-pushed on every connection, in that order.
- **The gantry is deaf while it moves.** `buildBlock()` runs homing, Z travel and the servo inside one synchronous function and never calls the serial reader. A build cannot be cancelled from software, and the interface is not allowed to imply otherwise.
- **No encoders anywhere.** Every stepper in the machine is open-loop. Position is a count, and its only anchor is a limit switch.

- **The Pi camera is reachable only through Picamera2.** The Pi 5's CSI camera has no usable V4L2 path for this sensor, so the vision code carries a Picamera2 backend as its primary and a V4L2 fallback for development on an ordinary machine.

2.3 Overall System Architecture

The system follows a strict master-slave architecture with one master and two independent slaves, chosen so that no part of the machine can move without the master having decided that it should. **A Raspberry Pi 5 (8 GB) is the sole master.** It owns the camera, the web server, the orchestration between the two boards and every safety rule in the system. **An Arduino MEGA 2560 runs the gantry** on its own USB serial link, and **an Arduino Uno runs the feeder** on a second, entirely separate USB serial link. The two Arduinos have no wire between them and neither can command the other; the Pi is the only thing that couples them.

That isolation is a deliberate safety property rather than a wiring convenience. A block is only allowed to be picked up because the Pi received one exact terminal message from the Uno saying a block is staged, and correlated it with the request it sent. If the boards could talk to each other, that permission could be granted somewhere the master cannot see.

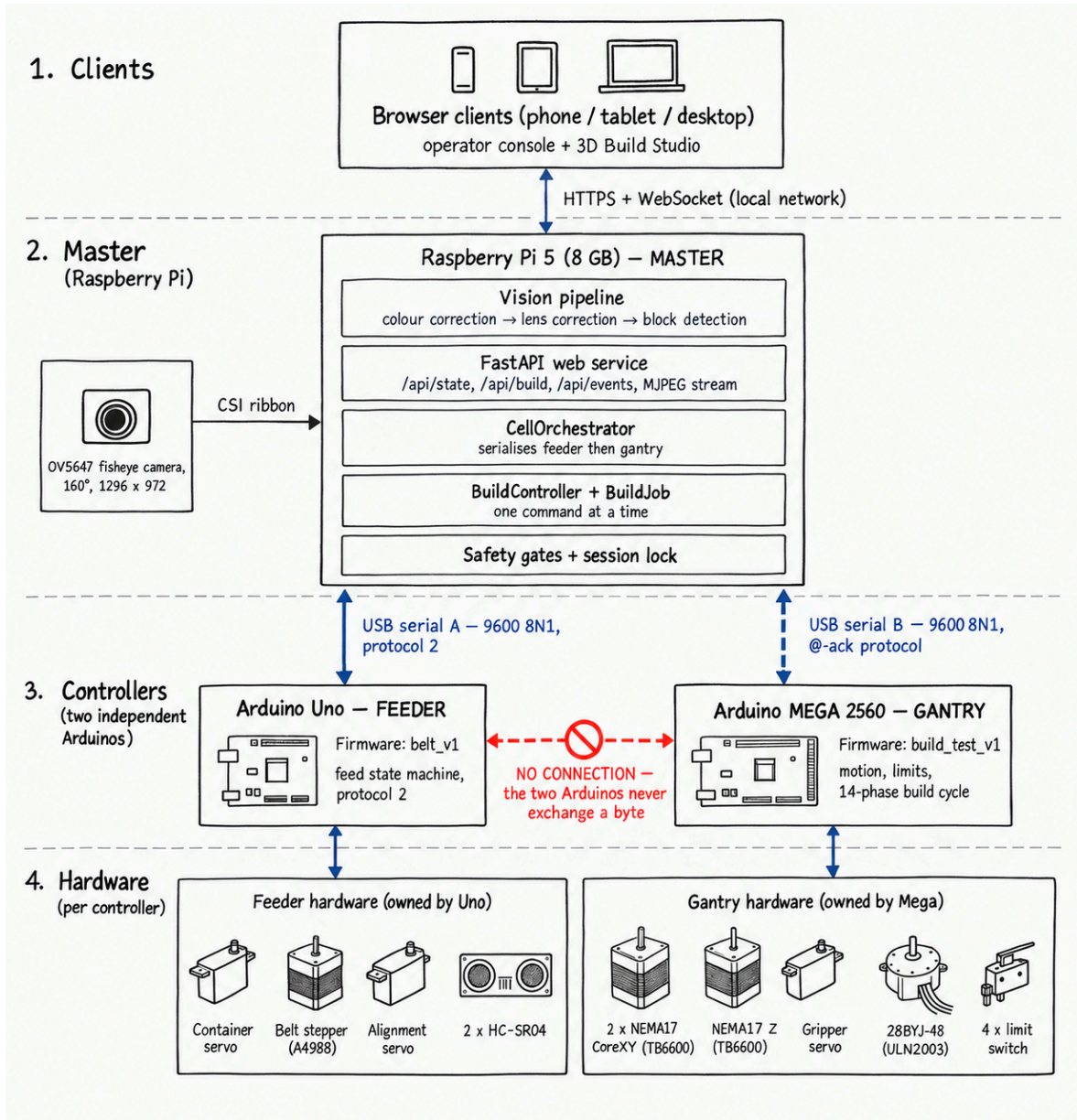


Figure 2: System block diagram: the three controllers, the two independent USB serial links, the camera, and the browser clients on the local network.

Raspberry Pi 5, 8 GB (the master): Camera capture and the whole vision pipeline, the FastAPI web service, the Studio, the two-board orchestration, and every safety gate and session lock. It talks to the browser over HTTPS and a WebSocket, and to the two boards over two USB serial links.

Arduino MEGA 2560 (the gantry): X, Y and Z motion, the claw servo, the rotation stepper, homing, limit enforcement, the grid geometry and the fourteen-phase build cycle. It answers on its own serial link at 9600 8N1 with @-prefixed acknowledgement lines.

Arduino Uno (the feeder): The hopper gate servo, the belt stepper, the alignment servo and the two ultrasonic sensors, together with the complete feed state machine. It answers on a second, entirely separate serial link at 9600 8N1, speaking protocol 2.

The vision system: An OV5647 fisheye camera on the Pi's CSI bus. Colour correction, lens correction, block detection, grid calibration and the pixel-to-cell mapping all run in-process on the Pi.

The operator interface: Any browser on the local network: the live camera view, cell selection, build confirmation, the 3D Studio and the digital twin, fed by REST calls and a durable WebSocket event stream.

The browser is treated as an untrusted mirror throughout. It sends requests and renders whatever the server reports; every guard is re-checked on the Pi, and a disabled button in the interface is never the thing that keeps the machine safe.

2.4 Mechanical Design

2.4.1 Overall structure

The machine is a rectangular aluminium-profile frame carrying a CoreXY gantry over a flat build surface, with a vertical Z column on the moving carriage, a feeder module standing alongside the frame on the left, and a camera looking straight down from a wooden support structure above. Every custom part is 3D-printed and every structural member is aluminium profile; nothing in the moving assembly is an off-the-shelf mechanism.

- **Holder travel, X: 22.8 cm = 4,550 steps.** The displacement of the holder reference from the X home switch to the X software cap. This is the number the grid is built on.
- **Holder travel, Y: 38.0 cm = 7,600 steps.** The same, for Y.
- **Z travel: 26.5 cm = 1,350 steps,** bottom switch to top switch, both physical.
- **Observed build footprint: 24.3 x 43 cm.** A separately measured observation of the surface the machine can reach. It is a record only: it does not replace the travel cap, and the extra reach is not modelled.
- **Wooden platform: 80 x 60 cm,** the base the gantry assembly stands on.
- **Feeder module: approximately 100 x 40 cm,** standing to the left of the gantry and feeding directly to cell [0,0]. The conveyor belt itself is about 30 x 10 cm of that, as a record.

2.4.2 The X/Y stage: CoreXY

The X/Y stage is a CoreXY (H-bot) arrangement carried over from the reference design. Two NEMA17 motors are fixed to the frame and drive one continuous GT2 belt path through a set of pulleys, so that neither motor rides on the moving beam. Motion is the sum and difference of the two shafts instead of one motor per axis:

Table 1: CoreXY motor directions, verified on the machine by watching the shafts.

Motor 1	Motor 2	Resulting motion	End of travel
CW	CW	X- (short axis, away from the X switch)	software cap at 4,550 steps
CCW	CCW	X+ (toward the X switch)	physical switch, pin 30
CW	CCW	Y- (long axis, toward the Y switch)	physical switch, pin 31
CCW	CW	Y+ (away from the Y switch)	software cap at 7,600 steps

Motors turning in the same sense walk the carriage along X; opposed senses walk it along Y. The advantage over a plain Cartesian layout is exactly the one the reference design states: keeping both motors off the moving beam makes the beam lighter and leaves a larger working envelope inside a given frame. The cost is a more complicated belt path with more pulleys to align, and, in this build, a coupling that is easy to get backwards: both drivers turn clockwise on an active-high direction pin, which is not what either datasheet suggests and is the result of one driver's coil wiring being physically reversed and the other ending up the same way after the machine was rewired. That is recorded in the firmware next to the constant so it is never 'fixed' in software without checking the wiring first.

Both X/Y axes are guided by castor wheels running on the aluminium profile, eight of them mounted on 3D-printed carriages. The single 1.5 m x 15 mm linear rail and its linear motion bearing are not used for X or Y: they carry the **Z** axis. Homing is by limit switch, one per axis, and each switch zeroes its own axis the instant it trips, so the corner where both are pressed is machine position (0, 0).

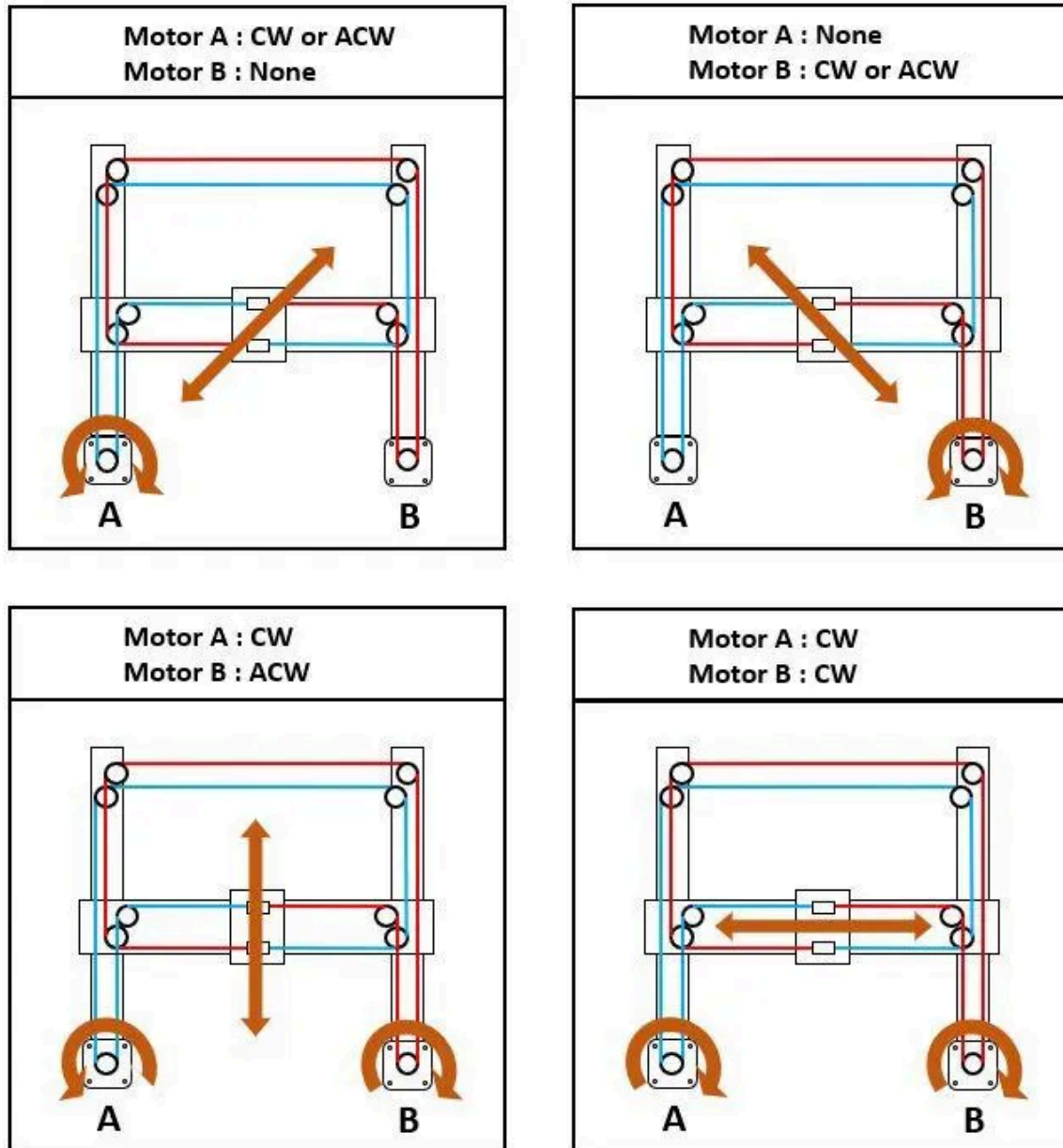


Figure 3: The X/Y stage: CoreXY belt routing, the two frame-mounted NEMA17 motors, the castor-wheel carriages and the two homing switches.

2.4.3 The Z axis

The Z axis is a single NEMA17 that is not coupled to the X/Y pair, carrying the claw assembly up and down a vertical column on the moving carriage. The carriage rides the 15 mm linear rail on its linear motion bearing, which is what gives Z its stiffness against the side load of a gripped block, and the motor drives it through a GT2 belt and pulley in the same family as the X/Y transmission.

Z is the only axis in the machine with a physical switch at **both** ends. The bottom switch on pin 28 is the ground reference the stack levels are measured from, and it is the axis' home: driving into it re-zeroes Z. The top switch on pin 29 is a far-end stop that halts the axis and reports where the top is without redefining zero. That end used to be a counted software limit, which only worked while the step count was trustworthy; replacing it with a switch means 'go to the top' became a seek rather than a move to a number, so it works even when Z has never been homed.

One consequence is worth stating because it is where the build cycle's reliability comes from: **every build re-establishes Z's zero on the physical ground switch before it picks a block up**, and carries the block at the full height of the top switch. Neither end of a build's Z travel is a remembered number.

2.4.4 The end effector

The reference design's end effector is an electromagnet under a board, dragging magnetised flat chess pieces across a plane. That is a good solution for chess and no solution at all here. In contrast, our blocks are plain wood, so there is nothing to attract; they are three-dimensional and have to be lifted and not dragged; and a magnet has no way to control which way round a piece ends up, which a machine with two block orientations needs. The end effector is a mechanical claw instead.

What it is: A 3D-printed two-jaw gripper driven by a single hobby servo on Mega pin 6, with exactly two commanded positions: OPEN at 0 degrees and CLOSE at 52 degrees. The jaws close on the middle of the block, across its 2.2 cm face.

Rotation: A 28BYJ-48 stepper through a ULN2003 driver turns the whole claw a quarter turn, 512 of its 2,048 steps per output revolution, at 10 rpm. That is what gives the machine its second block orientation.

What it does not have: Any sensor at all. The servo is commanded and then forgotten; nothing reports when the jaws have arrived, which is why every open and close inside a build waits a fixed 600 ms settle time before Z is allowed to move. The claw's rotation angle is likewise not sensed: the firmware tracks it relative to an assumed neutral start and the operator is trusted to begin with the claw physically neutral.

Why that is acceptable: The build cycle returns the claw to neutral at the feeder before every pickup and again after every placement, so a tracked angle is only ever one cycle old. A manual jog to an arbitrary angle is explicitly marked uncalibrated and blocks the **G** and mode-latch commands until a build returns the claw to neutral.

A mechanical grip introduced one geometric problem the electromagnet never had. The claw closes on the block's middle, but that middle does not sit on the rotation stepper's axis: it is offset by roughly $(-0.3, +0.6)$ cm. A 90-degree turn swings the block's centre **around** that axis rather than spinning it in place, and the swing is a constant $X +0.9$ cm, $Y -0.3$ cm. That is corrected as a tool offset in Section 3.2.3, and correcting it in the wrong place was one of the real errors of this project, described in Section 5.6.

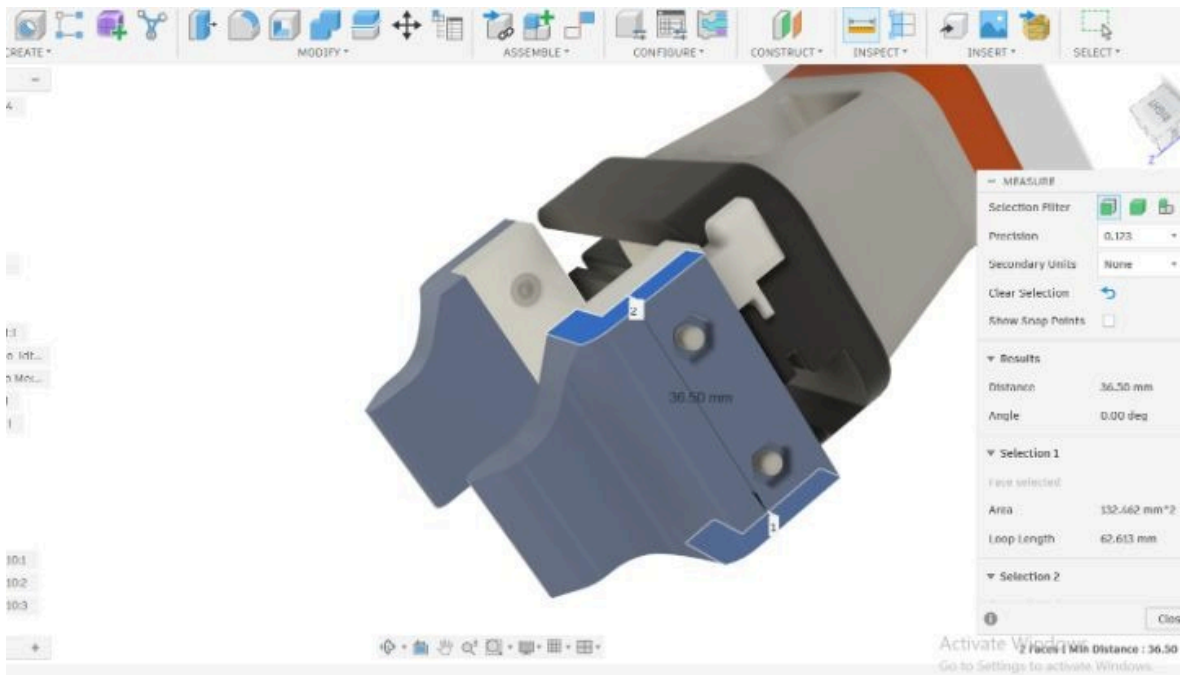


Figure 4: The claw: 3D-printed jaws, the gripper servo, and the 28BYJ-48 rotation stepper that gives the second block orientation.

2.4.5 The feeder module

The feeder is a self-contained module that stands to the left of the gantry frame and delivers a block to the gantry's cell [0,0]. It exists because the reference design does not need one: chess pieces are already on the board. A stacking machine consumes its parts, so something has to supply them, one at a time, in a known place and a known orientation.

Physically it is four things in a line:

1. **The container.** A hopper sized for the block footprint, with vertical elevation so a column of blocks can queue inside it, closed by a servo-driven gate on Uno pin 12. The gate does not simply open: it moves in two deliberate stages, 20 degrees closed to 90 degrees, then 90 to 160 degrees, with a 500 ms settle at each stage. Opening in one large movement releases blocks in a clump; opening in two lets the column settle against the gate and release the bottom block on its own.
2. **The exit sensor.** An HC-SR04 at the container's exit (TRIG 4, ECHO 5) that confirms a block has physically left the container and is on the belt. As soon as it fires the container is shut again, so a second block cannot follow the first out.
3. **The belt.** A GT2-driven conveyor about 30 x 10 cm, built from a belt sheet on 3D-printed rollers and driven by a NEMA17 through an A4988 (DIR 2, STEP 3), running at a configurable 150 steps/s by default. The belt carries the block from the container toward the pickup point.
4. **The stage sensor and the aligner.** A second HC-SR04 at the pickup point (TRIG 8, ECHO 9) stops the belt the moment the block arrives, and an alignment servo on pin 6 nudges the block square, moving from its 90-degree rest to 120 degrees and back after 350 ms. The stage sensor is then read again: only if the block is still there does the feeder report success.

The pickup point is the gantry's vertical cell [0,0], whose centre is the machine's home corner. Because the lattice is anchored on that corner, a pickup is a plain home with no move afterwards, and the claw closes on the block's centre without any additional positioning. The feeder never rotates: it always presents a block standing, whichever grid mode the gantry is latched into.

2.4.6 The workpiece and the build surface

The workpiece is a plain wooden block measuring **6.0 x 2.2 x 1.5 cm**, and the supply is **30 blocks, all of them bare wood in one uniform colour**. Nothing is painted, so the vision system has no colour signal to key on and has to separate the blocks from the work surface on tone alone, which is what Section 4.3.2's red-minus-blue segmentation exists to do. Twenty-nine of the thirty appear in the vision reference board used throughout Chapter 5; the thirtieth is the one being carried.

The build surface is the flat holder under the gantry, and it is registered to the machine by the machine itself and not by any fixture: **home is defined physically by where the X and Y limit switches are**, and every grid coordinate is measured from that corner. There is no separate datum to align and nothing to re-register if the surface is disturbed, provided the switches have not moved.

2.4.7 The two grids

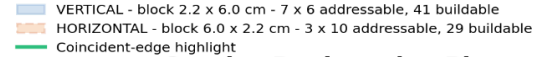
Because the block can be laid either way round, and because which way round it is laid changes how many cells fit and where they sit, this machine does not have one grid with an orientation flag. It has **two complete grids**, each with its own geometry, its own count and its own calibration:

Table 2: The two grids, at the shipped calibration. Gaps are a uniform 1.6 cm on every axis of both modes; pitch is block plus gap.

Mode	Axis	Block	Gap	Pitch	Cells	Cell centres (cm)	Block edges (cm)
vertical	X	2.2	1.6	3.8	7	0.00 to 22.80	-1.10 to 23.90
vertical	Y	6.0	1.6	7.6	6	0.00 to 38.00	-3.00 to 41.00
horizontal	X	6.0	1.6	7.6	3	1.90 to 17.10	-1.10 to 20.10
horizontal	Y	2.2	1.6	3.8	10	1.90 to 36.10	0.80 to 37.20

The vertical grid fills its travel exactly on both axes ($6 \times 3.8 = 22.8$ and $5 \times 7.6 = 38.0$), which is not a coincidence to be tuned away: it is what 'the build area is the travel area' means. Both grids share the same feeder cell at [0,0] and neither builds on it, so the buildable counts are $7 \times 6 - 1 = \mathbf{41 \text{ cells}}$ vertical and $3 \times 10 - 1 = \mathbf{29 \text{ cells}}$ horizontal. The lattice mathematics, the +1.9 cm registration the horizontal grid carries, and why that registration is not a tool offset, are set out in Section 4.2.3.

Drawn on top of one another the two grids show what the mode latch actually changes. The two lattices share one physical envelope and one feeder cell, and a horizontal column spans two vertical columns plus the gap between them ($6.0 = 2.2 + 1.6 + 2.2$, so $7.6 = 2 \times 3.8$), which is why the same surface reads as 7×6 cells one way round and 3×10 the other.



3 x 10 lattice, sharing one envelope and one feeder cell at [0,0].

2.4.8 Mechanical problems encountered during construction

Three mechanical problems were significant enough to change the design, and all three were answered in firmware rather than by re-making a part.

The slanted X rail: The asymmetrically loaded arm holder leaves the X rail slightly out of square with Y, so a placement that involves X travel lands off along Y by an amount proportional to the column index, measured at about 0.1 cm per column and with no row dependence at all. It is corrected by a firmware nudge applied only to the build motion (Section 4.2.5).

The pickup-rotate swing: The claw's grip centre is offset from the rotation stepper's axis, so a 90-degree turn carries the block centre around that axis by X +0.9 cm, Y -0.3 cm. Horizontal placements landed 1.4 cm too far from the X home switch until this was modelled as a per-rotation tool offset rather than as a grid error.

Constant placement error from the tool itself: Beyond the two effects above, the claw geometry and the block's own surface finish left a residual constant offset in placement. The firmware carries a per-mode **error_offset** pair exactly for this, applied like a grid trim; both modes currently ship at zero because the two effects above accounted for what had been measured. The block texture itself was a smaller but real problem, in two places: a rough or slightly cupped face changes where the jaws bite, and the natural wood grain is close enough in colour to the pale work surface that the vision system cannot separate them on brightness. The first is absorbed by the constant offsets above; the second is why block detection segments on red-minus-blue and red-minus-green rather than on brightness (Section 4.3.2).

2.5 Electrical and Control Design

2.5.1 Power architecture

The machine runs from a single 12 V / 15 A switched-mode supply with one LM2596 adjustable buck converter taking a 5 V rail off it. The split follows the current draw: everything that moves a stepper runs from 12 V directly, and everything small and logic-level runs from the 5 V rail.

- **The 12 V rail**, straight off the supply, feeds the three TB6600 drivers (the two CoreXY NEMA17s and the Z NEMA17) and the A4988 driving the feeder belt motor.
- **The 5 V rail**, from the LM2596 buck converter, feeds the gripper servo, the container and alignment servos, the 28BYJ-48 rotation stepper through its ULN2003, both HC-SR04 ultrasonic sensors, and the A4988's logic and reference supply.
- **Both Arduinos are powered over USB from the Pi**, which is also how they communicate.

- **The Raspberry Pi runs from its own official USB-C supply**, deliberately not from the buck converter.
- **One common ground** is shared by the 12 V supply, the buck converter, both Arduinos, all four drivers and every sensor.

The assorted resistors and capacitors in the bill of materials are used around the A4988 belt driver, for its current-reference network and for local decoupling on the motor supply.

The supply is rated at **15 A**, which is generous for what this machine actually draws. A NEMA17 of the size used here takes on the order of 1.5 A per phase, the 28BYJ-48 through its ULN2003 takes well under 0.3 A, a hobby servo draws a few hundred milliamperes while it is moving and almost nothing once it has arrived, and the two ultrasonic sensors are a few milliamperes each. Adding the worst case of every one of those together still leaves most of the supply unused.

In practice the machine never comes close even to that sum, because **the build cycle is sequential by construction and very little runs at the same time**. The fourteen phases move one thing at a time: Z travels while X and Y are stopped, X and Y traverse while Z is parked at the top switch, the gripper servo moves while nothing else does, and the rotation stepper turns only between a completed move and the next one. The feeder belt runs only while the gantry is idle waiting for a block. The instantaneous draw is usually one stepper plus the logic, so the 15 A rating is sized for the whole machine and not for any moment it actually reaches.

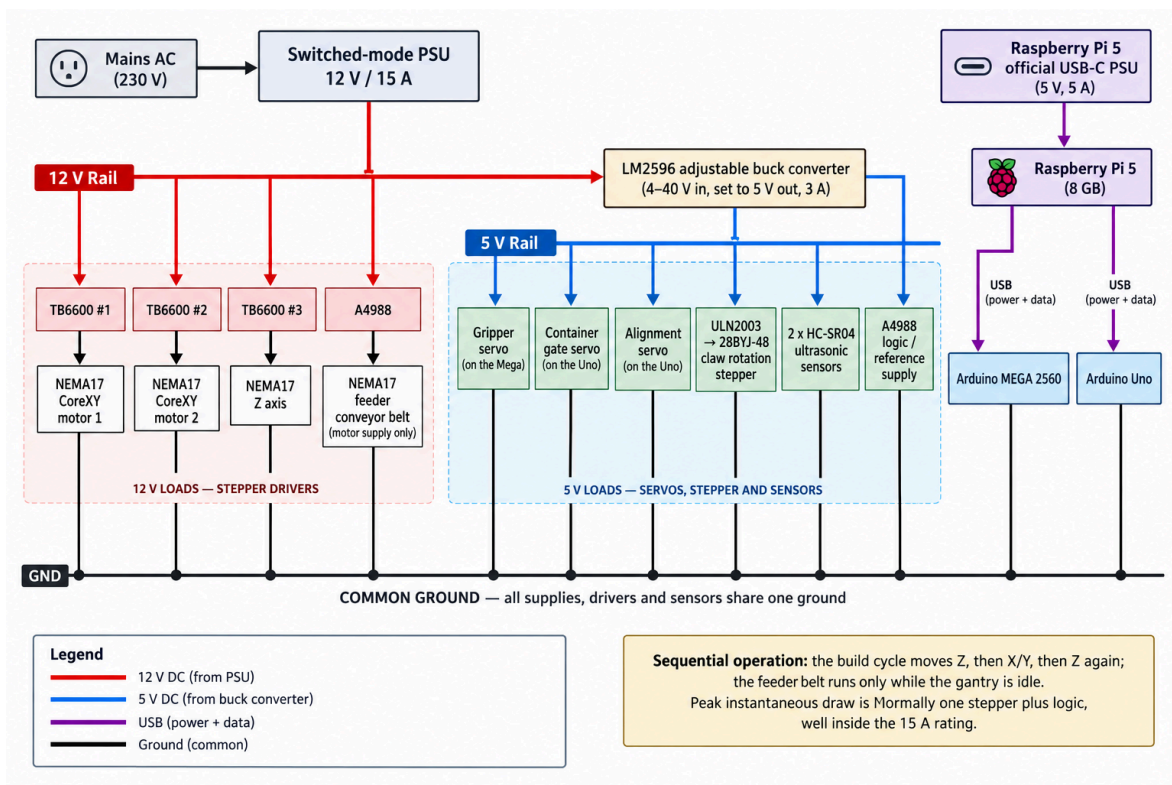


Figure 6: Power distribution schematic: the 12 V supply, the LM2596 buck converter, the two rails and the common ground.

2.5.2 Wiring and construction

Motor and sensor runs are made with connectors instead of being soldered directly to the boards, so that a motor or a sensor can be swapped without a soldering iron and so the gantry can be dismantled for transport. The TB6600 drivers take their motor and signal wiring on screw terminals. The communication wiring between the boards, the drivers and the sensors is made from cropped male-to-male jumper leads with heat-shrink over each joint, which keeps the run lengths honest and stops a jumper working loose against the vibration of the belt. There is no enclosure and no cable chain; cable management on the moving carriage is by routing and strain relief only, and the drag of that cable run is one of the contributors to the X-rail skew described in Section 2.2.1.

2.5.3 Control architecture

Control is hierarchical, and the division between the two levels is strict: **the Pi decides what and whether; each Arduino decides how.**

High level (Raspberry Pi): Chooses the target cell, the stack level and the grid mode; sequences the feeder before the gantry; applies every safety gate; owns the session lock; and decides whether a command may be issued at all. It never sends a motor step. What it sends is **B col row level** or **FEED id**, and nothing more detailed than that.

Low level (Arduino MEGA, Arduino Uno): Turns those commands into motion. The Mega owns step generation, direction polarity, homing, limit enforcement, the grid-to-steps arithmetic, the servo and stepper timing and the phase sequence. The Uno owns the feed state machine, the servo angles, the belt rate and the ultrasonic thresholds.

The reason for the split: The firmware owns everything that cannot change without reflashing, which is exactly the set of numbers that would be dangerous if the Pi held a stale copy: the step caps, the Z calibration, the pin assignments, the servo angles and the direction polarity. The configuration file owns what can change without reflashing: the grid counts, the block and gap geometry, the trims, the serial ports.

One consequence of that split is worth stating on its own, because it is the rule that keeps the two machines honest with each other. **A handful of physical values genuinely exist twice**, once in the Pi's configuration file and once compiled into the sketch, because the Arduino has no filesystem and cannot read the configuration at all. Every one of those pairs is listed explicitly in the repository's contributor notes and checked by an automated test that **parses the live sketch** and fails if either side has moved. A value that lives on two machines will eventually drift on one of them, and that test is the only mechanism in the project that makes the drift loud instead of silent.

The two firmwares are isolated from each other by construction. There is no wire between the Mega and the Uno, neither knows the other exists, and the only thing that couples them is the Pi's orchestrator. The permission to move the gantry is one exact message, **@id OK state=block_ready result=staged**, whose **id** matches the request the Pi sent; a terminal message with the wrong id is ignored, and every other message the Uno emits is progress telemetry that never counts as success.

2.6 System Workflow

The end-to-end sequence, from a structure that exists only as an idea to a finished stack, runs in five stages.

1. **Design.** A person opens the 3D Build Studio in a browser, places blocks in a live 3D scene on the machine's real lattice, and gets immediate feedback on whether each block is supported, whether it collides with another, and whether the structure would topple. The design is saved to the browser's own library as a **rigmodel/1** file.
2. **Compile.** The Studio turns the model into an ordered command program. The compiler builds a support graph from footprint overlap, sorts it bottom-up with Kahn's algorithm, groups blocks of the same orientation together to minimise grid-mode changes, and emits a deterministic list of **B col row level** commands separated by **R** and **RR** mode latches. An invalid model compiles to nothing at all rather than to a half-program.
3. **Feed.** For each block in the program, the Pi sends **FEED <id>** to the Uno. The Uno closes the container, opens it in two stages, waits for the exit sensor to confirm a block has left, shuts the gate behind it, runs the belt, stops on the stage sensor, nudges the block square, re-reads the stage sensor, and returns exactly one terminal result.
4. **Place.** Only on the Uno's correlated terminal success does the Pi send **B col row level** to the Mega. The Mega runs its fourteen-phase cycle: raise clear, home to the feeder, neutralise the claw, open, descend to the ground switch, grip, lift, traverse to the target cell, apply the grid's rotation, descend to the target level, release, then park by raising Z, homing X/Y and un-rotating the claw. It narrates every phase back over serial before that phase runs.
5. **Verify and repeat.** The overhead camera watches the surface throughout and the digital twin mirrors the build live from the phase telemetry. The runner waits for the durable terminal result before dispatching the next block, and the loop repeats until the structure is complete or something refuses.

The Studio is the normal entry point for building a structure. The console's click-to-build path is the same guarded operation with a single cell instead of a program, and is what is used for commissioning, for calibration and for placing one block by hand.

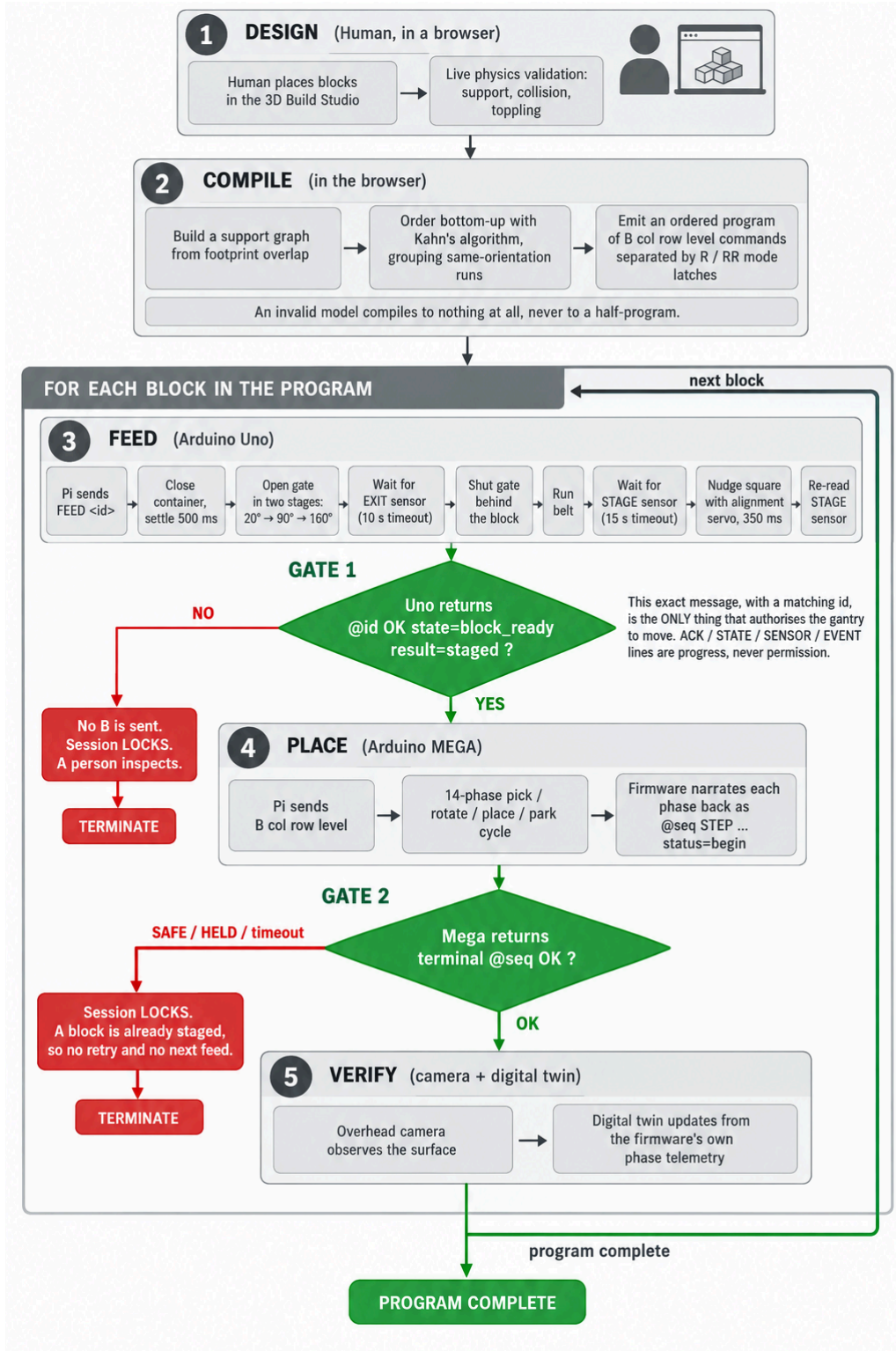


Figure 7: End-to-end workflow, from a design in the browser to a block on the stack, showing where each of the five stages is confirmed and not assumed.

3. Hardware and System Implementation

3.1 Cartesian Motion System

3.1.1 Motors and drivers

Three NEMA17 stepper motors move the gantry: two frame-mounted motors driving the CoreXY belt path, and one motor on the moving carriage driving Z. A fourth NEMA17 drives the feeder belt and belongs to the Uno and not to the gantry. Each of the three gantry motors has its own TB6600 driver, and each TB6600 takes a direction pulse and a step pulse from the Mega.

Table 3: Motor and driver inventory.

Motor	Type	Driver	Controller	Drives
Motor 1	NEMA17, 1.8 deg/step (200 steps/rev)	TB6600	Arduino MEGA	CoreXY belt, one of the coupled pair
Motor 2	NEMA17, 1.8 deg/step	TB6600	Arduino MEGA	CoreXY belt, the other of the pair
Motor 3	NEMA17, 1.8 deg/step	TB6600	Arduino MEGA	Z axis, independent of X/Y
Motor 4	NEMA17, 1.8 deg/step	A4988	Arduino Uno	Feeder conveyor belt
Rotation	28BYJ-48, 5 V, 2,048 steps per output revolution	ULN2003	Arduino MEGA	90-degree claw rotation
Gripper	Hobby servo	direct PWM	Arduino MEGA	Claw jaws, OPEN / CLOSE
Container gate	Hobby servo	direct PWM	Arduino Uno	Two-stage hopper gate
Aligner	Hobby servo	direct PWM	Arduino Uno	Squares the staged block

The firmware records X as 4,550 steps over 22.8 cm and Y as 7,600 steps over 38.0 cm, which is 199.56 and 200.00 steps/cm. A 20-tooth GT2 pulley advances 40 mm per revolution, so a 200 step/rev motor at quarter-step microstepping gives 800 steps per revolution, which is exactly 20 steps/mm, or 200 steps/cm. Y agrees to four significant figures and X to 0.2 %. Z is recorded as 1,350 steps over 26.5 cm, which is 50.94 steps/cm, and the same 20-tooth pulley at **full step** gives 50.0 steps/cm.

3.1.2 Homing and limit hardware

There are four limit switches, all plain micro switches, all wired normally closed and read through the AVR's internal pull-ups. Every switch is described in one table in the firmware by which **end of which axis** it guards, which is what lets Z carry two of them without any special-case code.

Table 4: Limit switches.

Switch	Pin	Wiring	Role	What tripping it does
X	30	NC	Home / zero for X	Re-zeroes X. This is the X+ end; X then runs 0 to -4,550 steps.
Y	31	NC	Home / zero for Y	Re-zeroes Y. This is the Y- end; Y then runs 0 to +7,600 steps.
Z bottom	28	NC	Home / zero for Z, and the GROUND reference	Re-zeroes Z. Every stack level is measured from here.
Z top	29	NC	Far-end stop for Z	Stops the axis and reports where the top is. It does not redefine zero.

Homing drives an axis into its own switch in 200-step chunks, with the switch re-checked on every step, and gives up if the axis has travelled more than twice its software travel plus 500 steps of slack, which is derived from the travel cap and not hard-coded so that re-tuning a cap cannot leave a stale homing limit behind. **Home is the corner where both the X and the Y switch are pressed**, and that corner is machine position (0, 0). The **0** command homes X and Y; **0+** additionally drives Z to its bottom switch and then seeks its top switch, which puts the whole rig into a known state and, on the way, measures the real switch-to-switch Z distance.

The important operational property is that **the go-to sequence re-homes every time**: home Y into its switch, home X into its switch, then move Y to the target row and X to the target column. A single move costs a full return to the origin, and in exchange no step error can survive from one move to the next. On an open-loop machine with no encoders that is the whole of the position-integrity story, and Section 5.2 shows what it costs in cycle time.

3.1.3 Software limits

X and Y are capped in software at the far end of their travel, because there is no switch there: X at 4,550 steps from its switch, Y at 7,600. A soft limit only ever guards the end an axis travels **toward**, which is always the end away from its home switch, so by construction there is exactly one per axis. Z has no software limit at all any more; its far end is the physical switch on pin 29, and the 1,350-step constant survives only as the calibration number that converts steps to centimetres. Getting that number wrong no longer risks a crash, because the switch stops the axis either way; it only skews the scale, which shows up as blocks landing slightly high or low.

3.2 End-Effector and Manipulation System

3.2.1 The gripper

What it does: Holds one block through the carry, the traverse and the descent, and releases it on command.

Inputs: One PWM channel on Mega pin 6. Two commanded positions only: **O** opens to 0 degrees, **C** closes to 52 degrees. **V <angle>** sets an arbitrary angle from 0 to 180 for bench work.

Outputs: None. The servo is commanded and forgotten; nothing reports when the jaws have arrived.

Typical workflow: Inside a build the jaws open at phase 4 while the claw is still clear above the feeder, close at phase 6 once Z is on the ground switch, and open again at phase 11 once Z is at the target level.

Validation and constraints: Because nothing reports arrival, every open and close inside a build is followed by a fixed 600 ms settle before Z is allowed to move. This is the single largest fixed overhead in the cycle after the Z travel itself, and Section 5.2.1 measures it at 0.93 s for an open and 0.95 s for a close including the phase pause.

Grip force is set purely by the closed angle, which was tuned on the rig until the jaws held a block without marking it. Over the sixteen logged hardware builds and the commissioning runs that preceded them, **no block was dropped**, crushed or lost from the claw during a carry.

3.2.2 The rotation stepper

A 28BYJ-48 through a ULN2003 turns the claw a quarter of its 2,048-step output revolution, 512 steps, at 10 rpm. The four ULN2003 inputs go to Mega pins 38, 36, 39 and 37, and the stepper library is driven in the IN1/IN3/IN2/IN4 order these boards actually require and not in pin order.

The mechanism has neither a home switch nor an angle sensor, so its state is tracked and not measured. Three things keep that tracking honest:

- **The build always returns the claw to neutral over the feeder before it descends** (phase 3), and again after the block is placed (phase 14). The tracked angle is never more than one cycle old.
- **A manual jog is marked uncalibrated.** A **<degrees>** is a signed relative jog capped at one turn per command, and an angle that is not exactly 0, +90 or -90 has no calibrated tool offset, so the **G** command and the grid-mode latches are refused until a build has returned the claw to neutral.
- **Rotation is a property of the grid, not of a block.** The **B** command takes no rotation word at all. Vertical mode places the block unrotated; horizontal mode turns it 90

degrees. A fourth word on a **B** command is a parse error whose message names the mode latch.

Rotation repeatability on the rig was good: the quarter turns land consistently and no cumulative drift was observed across a session. Neutral is re-established between sessions by the operator physically setting the claw square before power-on, which the firmware states loudly in its own startup banner because there is no way to verify it in software.

3.2.3 Tool-centre offsets

The X and Y counters describe the gantry **holder**, not the point where a block is actually released. The difference is a signed vector from the holder reference to the block-placement centre, and the firmware subtracts it from the requested cell centre before moving, so that the tool rather than its holder arrives at the cell:

```
holder target = desired block centre - tool offset(rotation)
```

neutral, (0.0, 0.0) cm: A vertical build never turns the claw, so the holder and the block centre coincide. This zero is genuine, not a placeholder.

CW (+90 degrees), (+0.9, -0.3) cm: The pickup-rotate swing. The grip point sits about (-0.3, +0.6) cm off the rotation stepper's axis, so a 90-degree turn carries the block centre round that axis instead of spinning it in place. This is the only rotation a build ever asks for.

CCW (-90 degrees), (0.0, 0.0) cm: No grid or build route requests a counter-clockwise placement, so this has never been measured. It is kept at zero, and kept in the list at all, because the offset lookup is written over all three rotation states.

The CW pair was derived from a rig measurement rather than from a drawing: a horizontal placement was landing 1.4 cm too far from the X home switch with Y dead on, and (+0.9, -0.3) is the unique swing that fits those two readings once the wrongly-signed grid offset it had been hiding behind was removed. Section 5.6 tells that story properly, because it is the clearest example in the project of a correction applied to the wrong quantity.

3.3 Processing and Control Hardware

Raspberry Pi 5: 8 GB of RAM, a Debian-based operating system and its own USB-C supply. It owns camera capture through Picamera2, the whole vision pipeline, the FastAPI service and its WebSocket event stream, the browser application it serves, both serial clients, the two-board orchestrator, and every safety gate and session lock in the system.

Arduino MEGA 2560: An ATmega2560 with 8 KB of SRAM and 256 KB of flash, on USB serial at 9600. It owns X/Y/Z step generation and direction, homing, the hardware

and software limits, the grid-to-steps arithmetic, the gripper servo, the rotation stepper, the fourteen-phase build cycle and the acknowledgement protocol.

Arduino Uno: An ATmega328P with 2 KB of SRAM and 32 KB of flash, on its own USB serial link at 9600. It owns the feed state machine, the container and alignment servos, the belt driver, both HC-SR04 sensors and the correlated protocol-2 serial interface.

3.3.1 Why a MEGA for the gantry

The gantry controller is a MEGA rather than an Uno or a Nano for the plainest of reasons: **pin count**. The gantry alone needs three step-and-direction pairs and two driver enable lines (eight pins), four limit switches, one servo channel and four ULN2003 lines for the rotation stepper. That is seventeen I/O lines before anything is added, against fourteen digital pins on an Uno, and it leaves nothing for the commissioning headers and spare lines that a machine under construction genuinely needs. A MEGA has 54.

Memory turned out to matter as well, though it was not the reason for the choice. The sketch prints a long human-readable banner, a full machine report, a grid map and a calibration table, and on AVR every plain string literal handed to **Serial.print()** is copied into SRAM at boot. Written the naive way this sketch needed 9,443 bytes of SRAM, which is 115 % of the MEGA's 8 KB, on a board with four times an Uno's memory. Wrapping every literal in **F()** moved them into flash:

Table 5: Gantry firmware resource usage, before and after **F()**.

Resource	Naive	With F()	Capacity
SRAM	9,443 bytes (115 %)	2,099 bytes (26 %)	8 KB
Flash	39,158 bytes (15 %)	40,590 bytes (15 %)	256 KB

That measurement is why the serial protocol looks the way it does. Space-separated **key=value** fields with no JSON, no braces and no quotes are not a stylistic choice: on an 8 KB AVR at 9600 baud, braces and quotes cost SRAM and airtime for nothing.

3.4 Vision System

3.4.1 Camera hardware and mounting

Camera: A DORHEA Raspberry Pi camera module: OV5647 sensor, 5 MP, with a 160-degree fisheye lens, connected to the Pi 5 over the CSI ribbon.

Capture: 1296 x 972 through Picamera2, which is the binned full-field-of-view sensor mode. Capturing binned instead of full resolution keeps the whole workspace in frame and keeps the per-frame processing cost inside the console's budget.

Mounting: Rigidly fixed, pointing straight down, approximately 50 cm above the build surface, on a purpose-built wooden support structure that spans the rig like a ceiling.

Nothing about the mount is adjustable in normal use, which matters: every saved calibration embeds the camera's projection identity and is refused by every consumer if the lens, orientation or framing has changed since.

A 160-degree lens at 50 cm is what makes a 23 x 38 cm workspace fit in one frame from a structure that can actually be built out of timber over a table. The cost is severe barrel distortion and a strong colour cast, and the next two sections are what those cost in software.

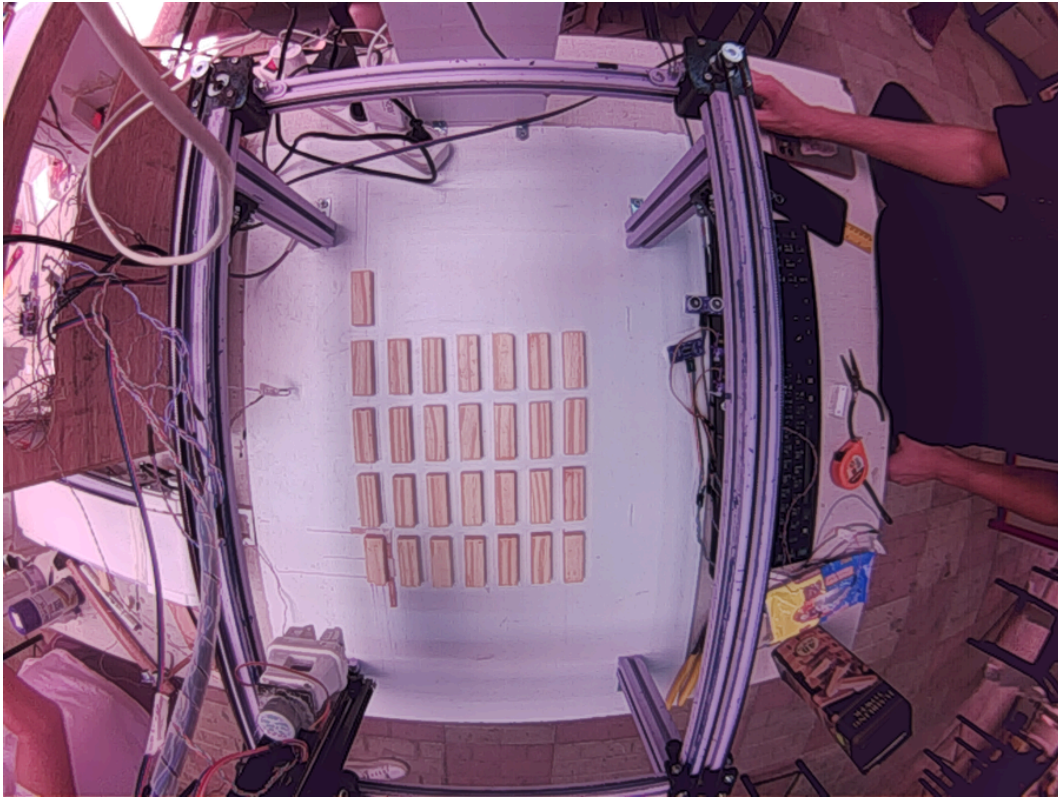


Figure 8: A raw frame straight off the camera, before any correction. The barrel distortion is visible in the aluminium rails, which are straight on the machine and curved in the image, and the magenta cast is the uncorrected colour response described in Section 3.4.3.

3.4.2 Lens correction

The fisheye is corrected by an equidistant projection model with polynomial radial terms, tangential terms and independent focal scales, all applied through a remap table built once at startup. The parameters actually in use on the delivered machine are:

Table 6: Lens correction parameters in `camera_settings.json`.

Parameter	Value	Note
<code>lens_fov_deg</code>	168.0	Diagonal reference. Tuned up from the advertised 160.
<code>model</code>	equidistant	The source-radius model.
<code>k1, k2, k3, k4</code>	0.14, 0.18, 0.035, 0.0	Radial terms.
<code>p1, p2</code>	-0.028, 0.0	Tangential terms.

Parameter	Value	Note
focal_x_scale, focal_y_scale	1.2, 1.0	Independent axis scaling.
skew	0.020	Small residual skew.
output_fov_deg	120.0	The corrected output field of view.
source	estimated	Not a checkerboard calibration.

3.4.3 Colour correction

The rig's camera has a colour cast strong enough to be a functional fault rather than a cosmetic one. The first live frame taken of the printed calibration sheet detected **nothing at all**, and the reason is worth recording because it was invisible from the outside:

Table 7: Measured on the first live frame: what the camera's white balance did to the scene.

Surface	HSV, raw	HSV, after white balance
Green ink on the sheet	(120, 49, 168)	(101, 48, 160)
Magenta ink on the sheet	(153, 97, 160)	(155, 75, 153)
White paper	(137, 37, 195)	(98, 15, 186)
The wall behind the rig	(139, 78, 176)	(135, 53, 168)

The green ink had moved to hue 120, which is cyan and not green, with a saturation of 49. It missed the green hue window and fell under the saturation floor at the same time, so half of every sheet was simply not in the image as far as the detector was concerned. The pink wall behind the rig, at saturation 78, landed **inside** the magenta window, so the frame was full of what looked like ink, none of it on the sheet. That same cast degrades the block detector, which keys on red-minus-blue.

Two corrections answer it, at two different levels. The sheet detector normalises every frame internally with a white-patch balance that scales each channel so the frame's 92nd-percentile brightness is neutral; it costs about 3 ms on a 1296 x 972 frame because it is a lookup table and not a float pass, and it no-ops on a frame that is already neutral. Above that, a saved colour transform is applied **once, at the captured frame**, immediately after frame orientation and before every detector, so that every tool in the system sees the same pixels. That transform is a 3 x 4 gain-plus-offset matrix with a saturation term, fitted by matching a camera frame against a phone photograph of the same scene; the fitted profile reached a similarity of **0.669** against a baseline of 0.404, short of its 0.95 target, and it is deliberately softened to 76 % strength to stop the offsets crushing the shadows.

Colour is explicitly **not** part of the saved projection identity, because recolouring changes no geometry and must not invalidate a saved calibration.

3.4.4 Calibration approach

Calibration here means establishing where the machine's grid sits on the camera image, which is a different question from correcting the lens. The result of a calibration is a saved **workspace_map.json** holding four workspace-envelope corners in normalised image coordinates, the physical grid geometry it was fitted against, and the camera projection identity it is only valid for. Three routes exist and all three write the identical artefact, so nothing downstream needs to know which one was used.

Three routes exist, and all three write the identical artefact:

Four clicked corners: The operator clicks the four corners of the holder envelope on the live image and the corners are saved directly. Built, and the fastest to run; this is the fallback.

Printed colour target: An A2 sheet carrying an 8 x 10 lattice of chromatic fiducials is laid on the surface with its lower-left corner on holder home. The detector segments the ink, assigns integer lattice indices by breadth-first walks, fits a homography from indices to cell centres, and re-scores every cell against the footprint that fit predicts for it. **This is the primary route, and the one presented here.**

Evidence-assisted printed target: The same sheet, but pooling accepted observations across several manually confirmed frames, so that cells hidden by the gantry in one frame can be measured in another. Built, for when the gantry occludes part of the sheet.

Placed-block self-calibration: The rig places its own blocks on commanded cells and the calibration is fitted to where they actually landed, so the thing measured is the thing being calibrated. Built, and the route with the strongest measured evidence (Section 5.4.2).

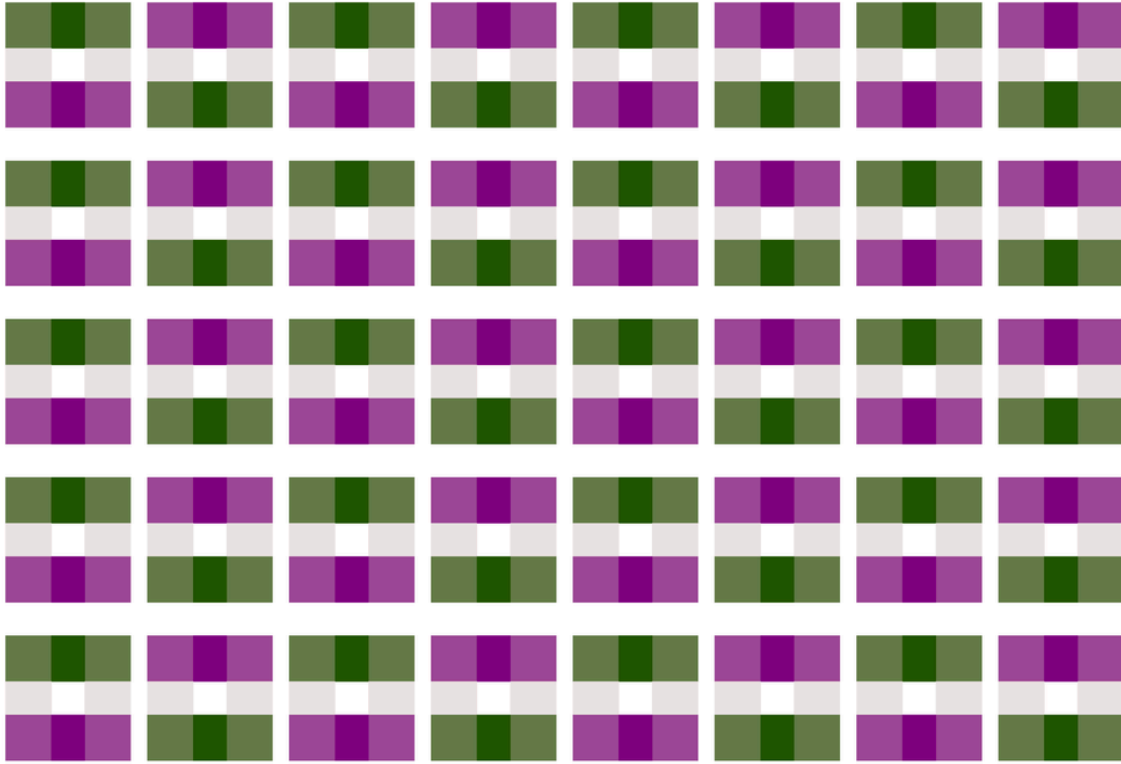


Figure 9: The printed A2 calibration target: an 8 x 10 lattice of green and magenta fiducial bars, printed at the rig's own cell geometry.

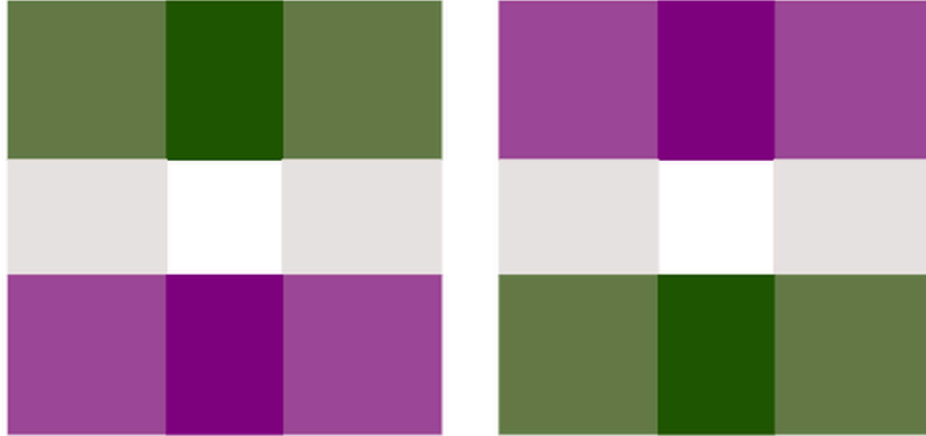


Figure 10: A detail of the same target. Each 6.0 cm chromatic bar reads as muted-dark-muted along X and is one vertical fiducial; the perpendicular 2.2 cm lanes read colour-beige-opposite colour and are the horizontal fiducials.

The physical sheet is A2 landscape, 59.4 x 42.0 cm, with 6.0 x 2.2 cm fiducial bars, 0.8 cm gaps along X and 1.6 cm gaps along Y, the lattice starting 0.8 cm from the page left and 4.8 cm from the page bottom. It is deliberately printed **larger** than the machine's grid so that it stays usable at any camera height without reprinting, which is also why choosing which mode-sized window of it to calibrate on is half the problem: the detector retains every strongly supported 7 x 6 or 3 x 10 window it finds, orders them by distance from the

image's bottom-left corner, and the operator selects one before saving. A window needs at least 95 % physical coverage with every row and column supported, and the fit is gated on colour parity, measured aspect ratio, mean residual and maximum residual as hard acceptance conditions rather than as reported statistics.

The sheet must be printed at 100 % scale, never 'fit to page', and laid so that the centre of its [0,0] block sits on the holder home point. Half of that block then hangs back past the switches, which is expected: the machine's own lattice is anchored the same way.

3.5 Sensors and Feedback

The complete sensor inventory of this machine is short, and its shortness is the point: everything the system knows about its own physical state comes from seven devices.

Table 8: Sensor inventory.

Sensor	Qty	Where	What it establishes	Signal
Micro limit switch	4	X+ end (pin 30), Y-end (pin 31), Z bottom (pin 28), Z top (pin 29)	The machine's origin, and the physical ends of Z travel. Three of them define a zero; the Z top one is a stop only.	NC contact to an internal pull-up, confirmed for 200 us before it is believed
HC-SR04 ultrasonic	1	Feeder container exit (TRIG 4, ECHO 5)	That exactly one block has physically left the hopper onto the belt.	Trigger pulse and echo width, 30 ms timeout, detection below 10.0 cm
HC-SR04 ultrasonic	1	Feeder pickup point (TRIG 8, ECHO 9)	That a block has arrived at the pickup point, and, read again after the aligner moves, that it is still there.	As above
Overhead camera	1	Approximately 50 cm above the build surface	Which blocks are on the surface, where, and what colour.	1296 x 972 frames over CSI through Picamera2

There are **no encoders anywhere in the machine**, and no current sensing on any driver. Every stepper is open-loop, so a mechanical stall produces step pulses that go nowhere and nothing reports it. The only thing that keeps position honest is the re-home before every move, and the only thing that would catch a stall is the camera noticing the block did not arrive, which is designed but not implemented (Section 5.6.2).

3.5.1 Ultrasonic signal acquisition

Each HC-SR04 is read by the standard method: hold the trigger low, pulse it high for 10 us, then measure the echo pulse width with a 30 ms timeout and convert at 343 m/s.

```

duration    = pulseIn(echoPin, HIGH, 30000);    // us, 0 on timeout
distance_cm = duration == 0 ? -1.0
              : duration * 0.0343f / 2.0f;
detected    = distance_cm >= 0.0f
              && distance_cm < DETECT_DISTANCE_CM;    // 10.0 cm

```

Two details in that are deliberate. A timeout returns -1 and is reported on the wire as **distance_cm=no_echo**, which is **never** treated as a detection: 'I heard nothing' and 'nothing is there' are different statements and collapsing them is how a feeder decides an empty belt is fine. And the sensors are sampled every 100 ms only while a feed cycle is actually running, with readings reported at cycle admission and on detections and not continuously, so the serial link is not flooded with polling.

The 10 cm threshold was set from the physical installation and proved reliable at both positions in bench testing, distinguishing a staged block from an empty belt without false detections from the belt surface or the container walls.

3.5.2 Limit switch acquisition

All four switches are normally closed to an internal pull-up, so a broken wire reads as a permanently tripped switch and stops the axis rather than letting it run into its end. A switch is polled on every single step and must read active continuously for 200 us before the firmware believes it, which is enough to reject contact bounce without adding a measurable delay to a 1.15 ms step. Trips are edge-counted per switch and not per axis, so that 'the Z switch was hit' is never an ambiguous statement, and the counts appear in the machine's full report.

3.6 Power and Safety

3.6.1 Power

The power architecture is described in Section 2.5.1: a 12 V / 15 A supply feeding the four stepper drivers directly, an LM2596 buck converter producing a 5 V rail for the three servos, the 28BYJ-48 through its ULN2003, both ultrasonic sensors and the A4988's logic supply, both Arduinos powered over USB from the Pi, the Pi on its own official supply, and one common ground across all of it.

The 15 A rating leaves the machine with comfortable headroom, because the four NEMA17 axes never all move at once: the build cycle moves Z, then X and Y, then Z again, and the feeder belt runs only while the gantry is idle. Even three axes simultaneously energised at a typical 1.5 A per phase is a small fraction of the supply. On a brown-out the failure mode would be a driver losing steps or an Arduino resetting mid-motion, and the second of those is detected: an unexpected **@0 BOOT** line during a session is treated as a hard error, because the claw may be holding a block.

3.6.2 Safety mechanisms that exist

1. **Hardware limit switches on every axis end that has one:** X home, Y home, Z bottom and Z top, all polled on every step and all able to stop a move.
2. **Software limits on the two ends that have no switch,** X- and Y+, checked before every move and not during it, so an out-of-envelope target is refused before anything turns.
3. **Pre-motion validation of every build.** The cell range, the grid readiness, the compensated holder target, both Z switches being enabled, and the Z calibration being sane are all checked before the machine moves at all; a failure at that stage is reported as **SAFE**, which means literally nothing moved.
4. **Two distinct failure kinds, never collapsed into one.** **SAFE** means nothing moved and a retry is fine. **HELD** means the failure happened part way through, the claw may still be gripping a block and the position is unknown, and it needs a person. They are separate message kinds precisely so that Pi-side code **cannot** handle them with a generic 'if not ok, retry'.
5. **Session lock on uncertainty.** An abort, a timeout, a board reset or a cable loss locks the session; every further mutation is refused and the only recovery is a person plus a service restart.
6. **The feeder refuses to double-load.** A **FEED** request arriving while the stage sensor already sees a block is refused outright with **stage_occupied**, before the container opens.
7. **One command at a time,** enforced at five independent layers from the HTTP route down to the serial client (Section 4.4.3).
8. **Cell [0,0] is never a build target,** refused in the firmware, on the Pi and in the browser.

3.7 System Integration

In the finished machine the electronics sit off the moving gantry entirely. The four stepper drivers and the buck converter are mounted on the frame beside the 12 V supply, with the two Arduinos alongside them; only the motors, the switches, the servo and the rotation stepper have wiring that travels with the carriage. The Raspberry Pi sits beside the rig with two USB leads running to the two Arduinos and the camera ribbon running up to the overhead mount.

The three integration rules that make the assembly work are:

1. **Each board is addressed by a stable device path, never by enumeration order.** A Mega enumerates as `/dev/ttyACM0` and a clone as `/dev/ttyUSB0`, and which board gets which number depends on plug order. The configuration names each

board's `/dev/serial/by-id/...` path instead, and the service refuses to start if the two entries name the same device.

2. **Each board's identity is validated on connect.** The Uno must announce `@0 READY firmware=belt_v1 protocol=2 board=uno` and it is matched field by field against the configuration; a mismatch closes the port and fails startup rather than proceeding with the wrong firmware. The Mega's handshake waits for its own `@0 READY grid=... mode=... sync` marker.
3. **Exactly one process owns each resource.** One camera object, one Mega serial port, one Uno serial port, all created once by the web service's lifespan. Running a commissioning script while the service is up will contend for the same board and reset it, which is documented as an operational rule because there is no OS-level exclusive lock enforcing it.

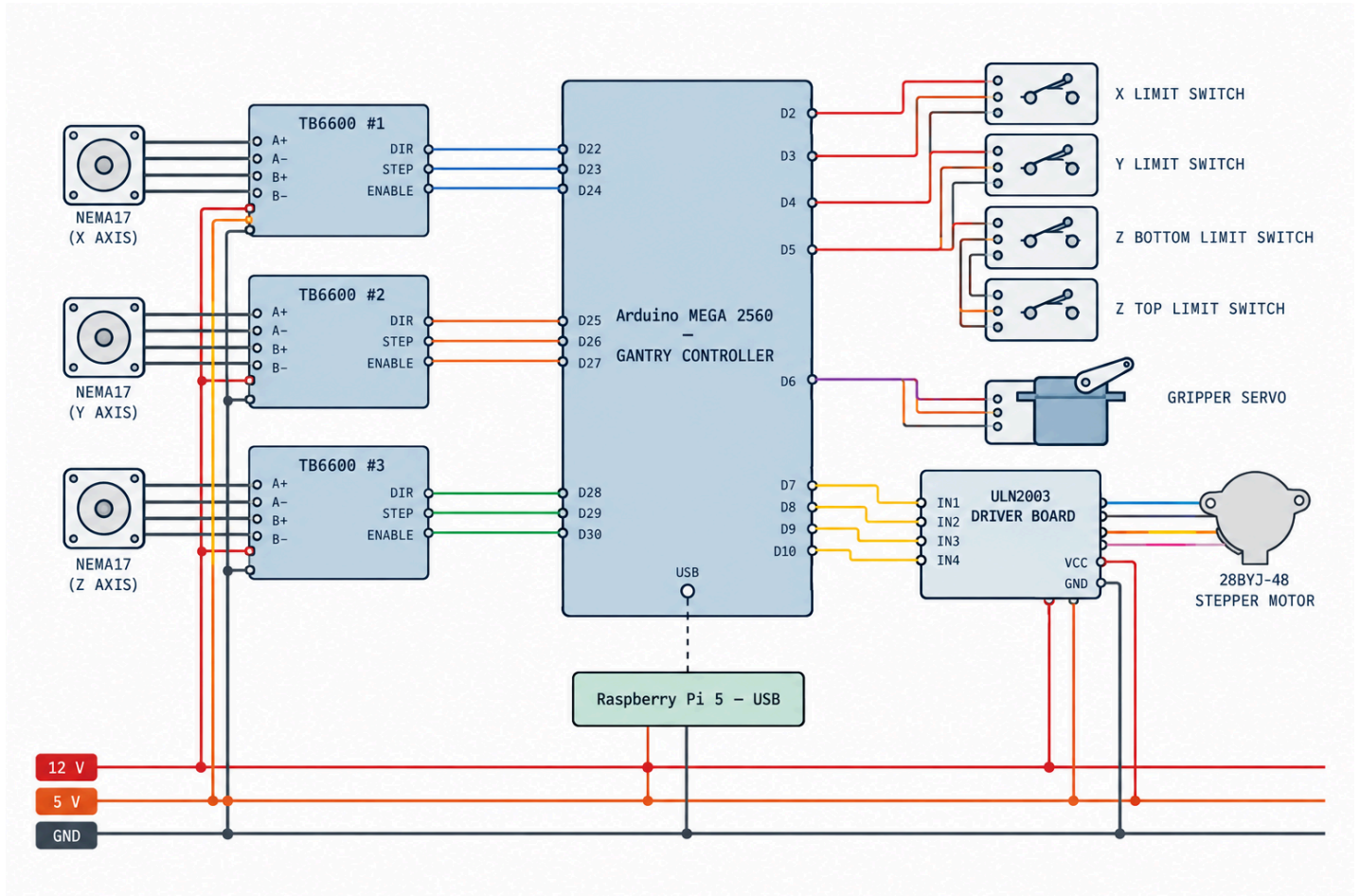


Figure 11: Complete wiring diagram: Arduino MEGA 2560 gantry controller, its three TB6600 drivers, the four limit switches, the gripper servo and the ULN2003 rotation stepper.

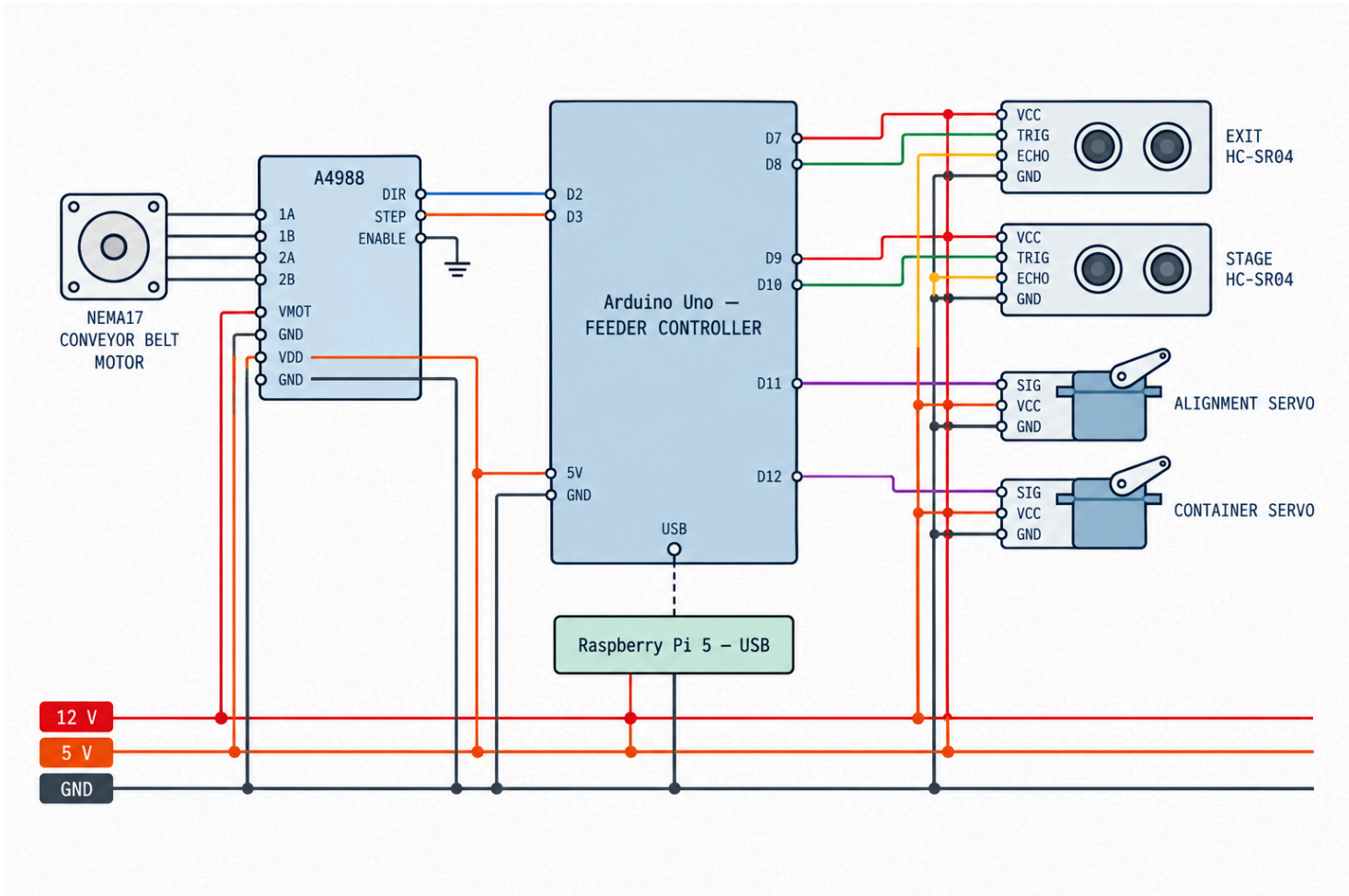


Figure 12: Complete wiring diagram: Arduino Uno feeder controller, the A4988 belt driver, the container and alignment servos, and the two HC-SR04 ultrasonic sensors.

The complete pin assignments for both controllers are tabulated in Appendix B, taken directly from the two firmware sources and not from any documentation of them.

4. Control and System Operation

4.1 Control Methodology

4.1.1 A hierarchy, and what closed-loop means here

The control system is hierarchical, with a hard boundary between the two levels. The Raspberry Pi decides **what** should happen and **whether** it is allowed to; each Arduino decides **how**. The Pi never sends a motor step, a direction bit or a millisecond delay: it sends **B col row level** or **FEED <id>**, and the firmware turns that into motion using numbers the Pi does not hold a copy of.

That boundary is drawn where it is because of one rule: **the firmware owns everything that cannot change without reflashing**. The step caps, the steps-per-centimetre ratios, the Z calibration, the block height, the build ceiling, the servo angles, the pin assignments and the motor direction polarity are all physical facts about the machine. Nothing can push them over serial, so a copy of them in the Pi's configuration would be a lie that nobody notices until the rig drives into something. What the configuration file does own is what can change without reflashing: the grid counts, the block and gap geometry, the trims, the two serial ports.

At the motor level the machine is **entirely open-loop**. There are no encoders, the step counters are trusted, and the only thing that keeps them honest is that every commanded move begins by driving both axes back into their home switches. When this report calls the system closed-loop, it means something specific and narrower: **sensor- and vision-confirmed sequencing**. The machine does not verify a shaft position; it verifies that a block left the container, that a block reached the pickup point, that a build phase began, that a block was released, and that a block is visible where one was commanded. Each of those is measured, and each of them can refuse permission for the step after it.

4.1.2 The acknowledgement protocol

The firmware talks to a human. Its serial output is a long, readable banner, a full machine report, an ASCII grid map and a running commentary on the build. Matching that prose is a bad way for the Pi to know what happened, because it breaks the day someone rewords a message, and because it cannot answer the two questions that matter during a forty-second build: is it still alive, and which of my commands is this about.

The answer was not to remove the prose. It is to print **one extra line beside it**, for the machine, on a channel a human can ignore by eye:

```
=====
BUILD COMPLETE - block placed at [3,5] level 0 (0.00 cm)      <- for the human
Place time: 31.6s
=====
@12 OK col=3 row=5 level=0                                   <- for the Pi
```

Every machine line starts with `@`, a character no other line in either sketch begins with, so the Pi's filter is one string test. The envelope is a sequence number, a kind, and space-separated **key=value** fields; there is no JSON, because on an 8 KB AVR at 9600 baud braces and quotes cost SRAM and airtime for nothing.

```
@<seq> <KIND> [reason text | key=value ...]
```

Table 9: Acknowledgement kinds emitted by the gantry firmware.

Kind	Terminal	Meaning
BOOT	no	@0 BOOT fw=build_test_v1 , the first line after a reset. An unexpected one mid-session means the board reset under the controller, which is a hard error.
READY	no	@0 READY grid=<cols>x<rows> mode=<mode> , the last line of startup and the Pi's sync marker. It carries the mode because a reset silently returns the board to vertical.
RCV	no	The command parsed and was accepted. It pins the sequence number to this command before the validation that may still reject it, which is what makes 'accepted, validating' a state distinct from 'moving'.
STEP	no	One build phase, announced before that phase runs.
ERR	yes	Refused on syntax or arguments. Nothing moved.
SAFE	yes	Refused on validation. Nothing moved . A retry is fine.
HELD	yes	Failed part way through. The claw may still be gripping a block and the position is unknown. Needs a person .
OK	yes	Finished: the block is placed and the rig is parked.

SAFE and **HELD** are separate **kinds** rather than a flag on one kind, and that is the single most important decision in the protocol. The firmware already draws the distinction for the human (one path prints 'Nothing moved', the other prints 'The claw may still be holding a block'), and it is the difference between a typo and somebody walking over to the rig. Making them two kinds means code on the Pi **cannot** collapse them into a generic **if not**

ok: retry; the **HELD** branch has to be written on purpose. The retry is the thing that breaks the machine.

The sequence number is assigned by the Arduino and not by the Pi, so that the command grammar stays typeable by hand in a serial monitor. Since the rig runs strictly one command at a time, 'the acknowledgement that followed my command' is never ambiguous. Section 5.6 records the honest weakness of that choice.

4.1.3 Real-time build progress

A build is about forty seconds during which the firmware never reads serial. Before the progress channel existed the Pi learned exactly two things in that window: that it had sent a command, and, eventually, how it ended. Everything in between was invisible, and every interface that wanted to show it had to invent something.

The firmware now prints one machine line per phase, immediately **before** that phase runs:

```
@12 STEP step=8 total=14 phase=move_to_target action=move \
    text=Move_XY_to_the_target_cell status=begin
[BUILD 8/14] Move X/Y to the target cell
```

Table 10: Fields on a STEP line.

Field	Meaning
<seq>	The same sequence number as the B it belongs to, so every phase is attributable to one command.
step	1 to total .
total	14, on the wire so that nothing downstream hard-codes it.
phase	A stable machine identifier that user interfaces switch on. Renaming one is a protocol change, not a wording change, and a silent one: a browser that does not recognise an id falls back to a generic 'moving' instead of crashing.
action	move / grip / release / rotate / park . Coarse on purpose: a consumer needs to know whether a block is being carried, not which motor turns.
text	The human label, underscored so it survives as one whitespace-separated token.
status	begin before the phase runs, and done exactly once.
ms	The firmware's own prediction of the phase duration. Z moves only , and omitted and not sent as zero when unknown.

Three properties of that channel are load-bearing. **One line per phase, never one per motor step**: fourteen lines is about 0.3 s of 9600-baud airtime inside a 40 s build, while per-step telemetry would be minutes of it, would fill the output buffer and would starve the terminal acknowledgement. **Exactly one done**, on phase 11, the instant the jaws open and the block is on the stack; nothing else can carry that fact, because parking is optional and there may be no phase 12 to imply it, and it is emphatically **not** terminal: the command is

still running and a parking failure downgrades the whole build to **HELD**. And **ms= is a floor, not a schedule**: nothing moves faster than its step rate, so the real phase can only take longer, and no consumer may treat its expiry as the phase having finished.

The reason the firmware sends **ms=** rather than the browser computing it is the hierarchy rule from Section 4.1.1. Working the descent out requires **Z_TRAVEL_STEPS**, **Z_TRAVEL_CM** and **BLOCK_HEIGHT_CM**, all three of which are firmware-only numbers, and a browser holding copies of them would drift silently the day the Z step rate is retuned. The board owns the numbers, so the board does the arithmetic. For the shipped calibration that is:

```
full travel      1350 steps      = 2565 ms  (+ DIR_SETTLE_MS = 2570)
one block height 76.4 steps      = 145 ms
descent to level K 1350 - 76.4*K steps = 2565 - 145*K ms
```

4.2 Motion Control

4.2.1 Command vocabulary

The gantry firmware's commands are the contract between the two machines. They are short because they are typed by hand during commissioning as often as they are sent by the Pi.

Table 11: The gantry (MEGA) command vocabulary.

Command	Does	Moves?
1 2 3 4	One manual jog of stepsPerMove on X- / X+ / Y- / Y+.	yes
D / U	One manual Z jog down / up.	yes
0	Home: drive into the X and Y switches. This is the origin.	yes
0+	Full reset: Z down to the bottom switch, Z up to the top switch, then home X and Y. It also measures the real switch-to-switch Z distance.	yes
G <col> <row>	Go to a cell centre. Both axes always move, after a re-home.	yes
B <col> <row> <level>	The complete fourteen-phase build cycle. Three numbers, no rotation word.	yes
O / C	Servo open / close.	the claw
V <angle>	Servo to an arbitrary angle, 0 to 180. Bench use.	the claw
A <degrees>	Signed relative rotation jog, -360 to +360, positive clockwise. Bench use; capped at one turn per command because the mechanism has no limit switch.	the claw

Command	Does	Moves?
R / RR	Latch the vertical / horizontal grid. Neither moves anything. Each is refused when it is already true, and both need X and Y homed first.	no
S <cols> <rows>	Set the highest addressable index for the active mode only. The other mode keeps what it was given.	no
shiftX <cm> / shiftY <cm>	Translate the whole placement lattice of the active mode. The pick-up is not shifted.	no
5 / 9 / Z / ?	Full machine report / ASCII grid map / Z calibration table / help.	no
6 / 7 / 8	Reset statistics / disable the X-Y drivers / declare the current position zero without homing.	no

Two of these changed meaning during the project and one lost an argument, and all three changes are worth stating because they are the kind that break a controller silently. **R** and **RR** used to be free rotation jogs and are now grid latches that move nothing. **A <degrees>** is manual bench rotation and explicitly not a new grid orientation. And **B** no longer takes a rotation word: **how the block is laid comes from the active grid, not from the command**, because a per-block rotation could place a rotated block inside a grid whose cell geometry does not match it, which is exactly the failure the two-grid design exists to prevent. A fourth word on a **B** is now a parse error whose message names the mode latch.

4.2.2 The coordinate system and homing

Each home switch zeroes its own axis the instant it trips, so the corner where both the X and Y switches are pressed is machine position (0, 0). The two switches sit at opposite ends of their axes, so the two axes do not share a sign:

```
X switch at the X+ end -> X runs 0 ... -4550 (software cap)
Y switch at the Y- end -> Y runs 0 ... +7600 (software cap)
Z switch at the Z- end -> Z runs 0 ... +1350 (TOP SWITCH, physical)
```

Grid indices hide that sign asymmetry entirely: column 0 is nearest the X switch, row 0 is nearest the Y switch, and both increase away from home. Every axis is described generically as extending from 0 in the direction of its travel end for its travel distance, whether that far end is held by a software cap (X, Y) or by a switch (Z), which is what lets one set of code handle all three.

The go-to sequence is deliberately expensive:

1. Home Y into its switch, then home X into its switch. That is the origin.
2. Move Y to the target row.
3. Move X to the target column.

Re-homing every time means lost steps never accumulate. It costs a full return to the origin on every move, and Section 5.2.2 measures that cost: it is the second-largest term in the cycle time, and it is the price of position integrity on a machine with no encoders.

4.2.3 The lattice, and why there are two grids

A block measures 2.2 x 6.0 cm in plan and can be laid either way round, and which way round it is laid decides how many cells fit, where they sit and how far the grid has to be registered. That is not one grid with a flag; it is two grids, each with its own complete geometry. Both share one physical envelope, because a block lying down does not move a limit switch.

The lattice itself is one line of arithmetic, and it is **centre-anchored**:

```
pitch      = block + gap
centre(i) = trim + error_offset + shift + i * pitch
```

Cell indices are 0-based and **coordinate zero is a real block whose centre sits on the home corner**. There is no leading gap, no trailing gap and no centring of the allocation inside the travel: the trim is the only thing that moves a grid. Two consequences follow. Cell 0's block hangs half a block back past the switches, which is expected and is what the per-mode edge-overhang budget exists to allow. And a full-travel grid lands its last centre exactly on the software cap, which is why the vertical grid's $6 \times 3.8 = 22.8$ cm and $5 \times 7.6 = 38.0$ cm come out even: that is what 'the build area is the travel area' means, and it is why vertical X has seven columns and not six.

The gaps are a uniform **1.6 cm on every axis of both modes**. That was settled by measuring the printed sheet (6.00 cm tiles, 1.56 cm gaps, identical on both axes); an earlier revision claimed 0.8 cm along Y, which made the horizontal Y lattice alternate 0.8 and 1.6, and that alternation was an artefact of the wrong gap and not a feature of the paper.

Cell [0,0] is the **feeder** in both modes and is never built on. The feeder never rotates: a block is always presented standing, on the vertical [0,0] footprint, whichever mode is latched. Because the lattice is centre-anchored, that cell's centre **is** the home corner, so a pick-up is a plain home with no move afterwards and the claw closes on the block's middle without any additional positioning. That costs exactly one cell per mode and no more: **B 0 0 <level>** is an inert no-op, while **B 0 3** and **B 4 0** are ordinary placements.

4.2.4 The horizontal registration, and four kinds of offset

The horizontal grid ships with a trim of **+1.9 cm on both axes**, and the reason is geometric rather than empirical. The block is picked up standing at the vertical [0,0] feeder, centred on home, and then rotated 90 degrees about the grip. The rotated 6.0 cm face overhangs the 2.2 cm vertical footprint by $6.0/2 - 2.2/2 = 1.9$ cm per side, so a +1.9 cm trim on each axis seats horizontal [0,0] flush against the vertical [0,0] block's edge (the near edge in X, the far edge in Y).

Four families of offset exist and each has exactly one job. Keeping them separate is what makes calibration falsifiable: two knobs that both look like 'shift everything' cannot be told apart by any measurement.

Table 12: The four offset families.

Family	What it is	Moves	Shipped value
block / gap	The physical lattice itself.	the cell spacing	2.2 / 6.0 cm and a uniform 1.6 cm
trim	Registration of a whole grid against the home switches.	every cell centre of that mode	vertical (0, 0); horizontal (+1.9, +1.9)
error_offset	The calibration knob: a constant per-mode nudge for a measured placement error. It cannot fix an error that grows with distance, which is a scale or pitch problem.	every cell centre of that mode	(0, 0) in both modes
tool_offset	Purely mechanical: how far the claw's grip centre sits from the holder centre, per rotation state.	the holder , not the cells	neutral (0, 0); CW (+0.9, -0.3)

Because a tool offset never enters the lattice, the cell centres in the table in Section 2.4.7 are also where a placed block physically comes to rest, which is why the camera overlay, the Studio and the digital twin can all draw the truth without carrying a correction of their own.

4.2.5 X-rail skew compensation

A placement made with pure Y motion (the same column, for example **B 0 3 0**) lands exactly where the grid says. A placement that involves X motion lands off along Y, and the error grows with how far along X the rig travels. It is not a constant offset.

Measured on the rig, the introduced Y error is **0.10 cm per column of X travel**, and it is linear in the column index with no row dependence at all: 0.00 cm at column 0 where there is no X travel, 0.10 cm at column 1, 0.20 cm at column 2, and $0.10 \times k$ cm at column k.

The cause is mechanical and is described in Section 2.2.1: the arm holder's asymmetric loading leaves the X rail very slightly out of square with Y, so the further the carriage is driven along X, the more of that angled rail it has travelled over and the more it drifts along Y. The drift is linear in the column index with no row dependence, which is exactly what an angled rail predicts and is a useful check that the diagnosis is right.

Re-machining or re-bracing the rail was out of scope, so the drift is cancelled in firmware. For a build the Y target is deliberately offset by exactly the drift the rail will add, so that the two cancel:

```
yNudge_cm = SKEW_Y_PER_COL_CM * col // 0.1, measured
            + SKEW_Y_PER_ROW_CM * row // 0.0, no row dependence
            + SKEW_Y_PER_COLROW_CM * col * row // 0.0, cross term, unused

targetY = cellTargetPosition(Axis_Y, row, rot) + lround(yNudge_cm * stepsPerCm_Y)
```

The scope of that correction is as important as the correction itself. It is applied in **gotoBuildTarget()** alone, and the **B** motion is the only path that gets it. It is not in the cell-centre arithmetic, not in the **G** command, not in the grid map, not in the Python link, and not in the camera grid, the Studio grid or the 3D lattice. **Every representation of the grid stays a perfectly rectangular, level lattice**; this bends only the physical motion, so that the real blocks come out straight. X is never touched, and the result is clamped to the Y travel so that a bad coefficient cannot drive the carriage past a soft limit.

The correction was flashed and tuned on the rig rather than left on paper: the serial log shows it running at 0.150 cm per column in the morning session of 3 September 2026 and re-tuned to **0.100 cm per column** for the afternoon session, with the per-cell correction logged as it is applied (**X-rail skew: Y 0 -> 20 steps (0.100 cm, col skew)**).

4.2.6 Z heights and block levels

Everything the build knows about height falls out of two measurements and is computed at run time and not hard-coded, so re-measuring the rig is a one-line change.

Table 13: The Z / block-level calibration.

Quantity	Value	Derived as
Z travel, steps	1,350	Bottom switch (pin 28) to top switch (pin 29). Calibration only; the switch is what stops the axis.
Z travel, cm	26.5	Tape-measured.
Scale	50.9434 steps/cm	1,350 / 26.5, computed at run time
Block height	1.5 cm = 76.42 steps	One stack level
Fixed margin	+0.10 cm	Added once to any level ≥ 1 . Level 0 ignores every margin, because ground is a physical switch and cannot drift.

Quantity	Value	Derived as
Per-level margin	0.00 cm	For an error that accumulates up the stack. Not needed at the shipped calibration.
Build ceiling	25.0 cm	Deliberately below the 26.5 cm travel, so there is always clearance to fly a block over the tallest possible tower.
Highest level	16 (24.0 cm)	floor(25.0 / 1.5), leaving 2.5 cm of headroom

```
target_cm    = level * (BLOCK_HEIGHT_CM + Z_MARGIN_PER_LEVEL_CM) + Z_MARGIN_FIXED_CM
target_steps = round(target_cm * zStepsPerCm()) + Z_MARGIN_FIXED_STEPS
```

The two margins answer two different errors and must not be confused. If a placed block ends up 1.52 cm high instead of 1.50, the error is **per level** and accumulates up the stack. If every level is uniformly a little low or high because of claw geometry, a switch tripping early or a lip on the block, that is a **constant** error. The shipped calibration carries +0.10 cm of fixed margin and no per-level term, which says that the block height itself is right and the whole stack sits slightly low.

4.3 Vision and Positioning

4.3.1 The pipeline, in order

One process owns the camera, and every frame passes through the same fixed sequence before anything is asked of it:

1. **Capture** at 1296 x 972 through Picamera2, into a non-blocking latest-frame pump so that a slow analysis can never stall the video.
2. **Frame orientation**: the configured flip and rotation, applied first so everything downstream shares one convention.
3. **Colour correction**: the saved gain/offset/saturation transform, applied once here and never inside an individual detector, so every tool in the system sees identical pixels (Section 3.4.3).
4. **Lens correction**: the fisheye remap table, followed by the framing crop and zoom (Section 3.4.2).
5. **Detection**, on a background worker at a lower rate than the video, with the main loop drawing the last completed result instead of waiting.
6. **Mapping**: the saved workspace homography turns a pixel into physical centimetres and then into a **[col, row]** cell.

The Pi picks the cell. **The firmware alone turns a cell into safe step targets**, and the Pi never needs a motor step to draw or select one.

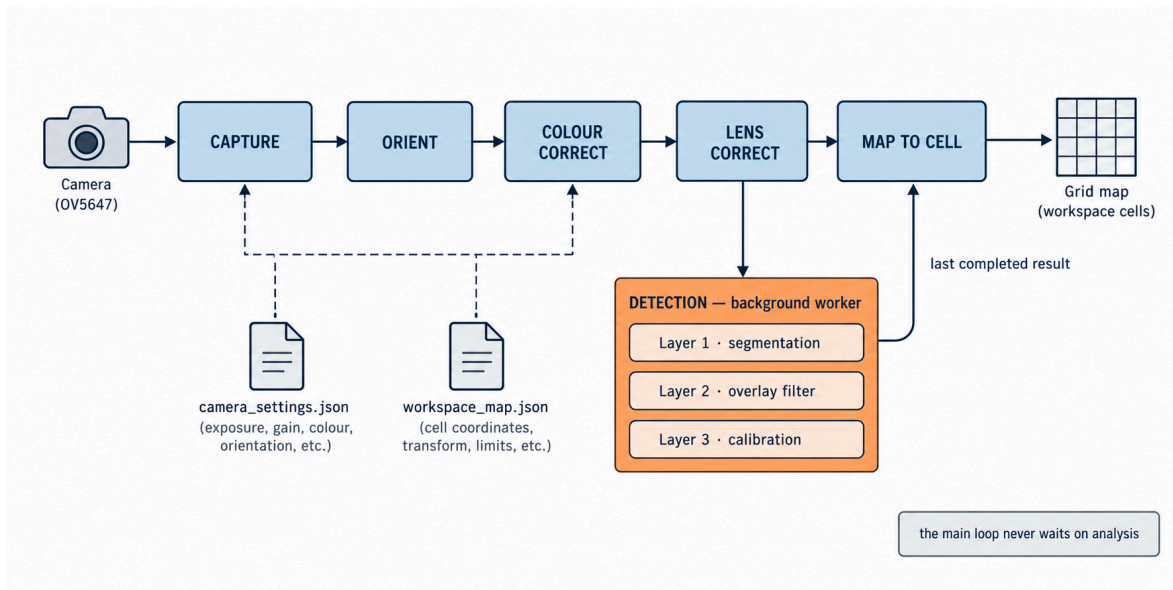


Figure 13: Vision pipeline: capture, orientation, colour correction, lens correction, detection, and the pixel-to-centimetre-to-cell mapping.

4.3.2 Block detection: three layers

There are three things in this system that find blocks. They share a segmentation front end and then diverge completely, and confusing them is the main way to get a wrong answer.

Table 14: The three detection layers.

Layer	Answers	Runs	May write a calibration
1. Segmentation	What warm, block-shaped things are in this frame?	every analysed frame	no
2. Live overlay	Which of those are really blocks, and where do I draw them?	every analysed frame	no
3. Calibration	Where is the machine's grid, in pixels?	once, deliberately	yes

Layer 1 segments warm material by **red-minus-blue** and **red-minus-green** exceeding thresholds, never by brightness: the work surface is overexposed in the captures this was tuned on, so a brightness cutoff selects the table and not the blocks. The mask is opened with a 3x3 kernel and closed with a 5x5, both deliberately small, because a large close joins neighbouring blocks together before the splitting step has a chance at them. A contour then becomes blocks by one of three paths, tried in order: straight through if it is already one standard-sized rectangle; a **touching-block split** that connects the two deepest convexity defects and cuts the component in two; or a **compound decomposition** that fits ideal block-sized rectangles into a merged component. What layer 1 returns is a hypothesis, not an answer, and it cannot be anything else: a rail and a block are the same shape.

Layer 2 is where a hypothesis becomes an overlay. It rejects, then rectifies: duplicate detections are collapsed by intersection-over-union (the compound decomposition proposes overlapping rectangles, so a block joined to its neighbour by a shadow yields a third rectangle straddling the seam); anything whose box runs off the frame is dropped, which is what removes the aluminium rails at the frame edge; anything further than 0.34 cells from an integer lattice site is dropped, which is what removes the holder's two thin wooden offcuts beside [0,0]; and every survivor is redrawn with the population's median size and the lattice's shared bearing.

Two rules keep layer 2 honest and both are asserted by tests. **The measured centre is never snapped to the lattice.** Sharing a size and a bearing is what makes a full board read as one grid; snapping the positions too would draw a prettier grid and hide a misplaced block, which is the one thing this overlay exists to show. And **the lattice filter must not fire on a partial view**: below six detections, or when the recovered lattice would reject more than 30 % of what it saw, everything is kept, because a lattice fitted to four blocks is not the board's lattice.

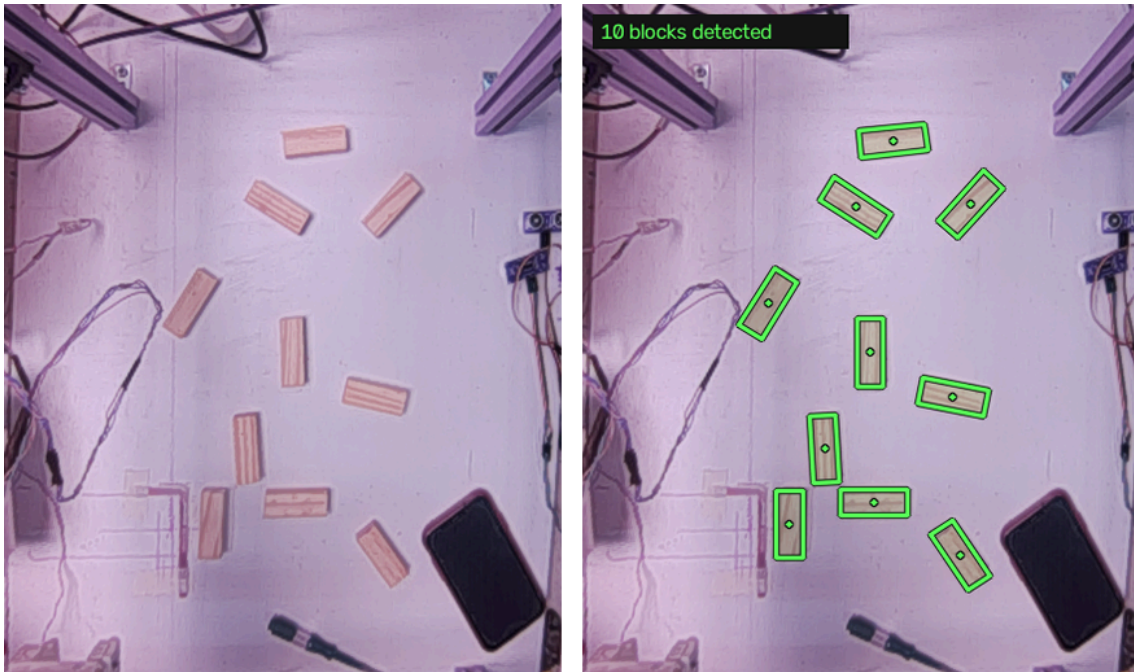


Figure 14: The detector on a real frame. Left: ten blocks scattered on the surface, deliberately off the lattice. Right: the same frame through layer 2, with all ten found and squared to a shared size, their measured centres marked. Nothing else in the frame is claimed as a block: the aluminium rails at the edges, the cable runs, the screwdriver and the phone are all rejected.

4.3.3 Grid calibration from the printed target

The printed A2 target described in Section 3.4.4 is the calibration route presented here. What follows is how the fit works, and every gate it has to pass.

What it does: Recovers the four corners of the machine's holder envelope in normalised image coordinates, so that a pixel can be converted to a physical position and then to a grid cell.

Inputs: One camera frame containing the printed sheet, laid with the centre of its [0,0] block on the holder home point; the active grid mode; and the block and gap geometry from the configuration.

Outputs: A `workspace_map.json` holding four normalised corners, the physical grid geometry the fit was made against, and the camera projection identity it is valid for.

Typical workflow:

1. The frame is white-balanced first, driving its bright quantile to neutral. Without this the green ink is not green (Section 3.4.3).
2. Two hue windows segment the two inks, on hue plus a saturation floor and not on brightness. The inks stay far apart in hue (green at 58 to 115 in OpenCV's 0 to 179 scale, magenta at 130 to 178, with nothing in between) and the saturation floor of 32 sits between the rig's green ink at 48 and its white paper at 15.
3. Each blob becomes a rotated rectangle. Broad aspect, area and colour-purity checks decide which rectangles may vote on the median size and the long-axis direction; rejected blobs are still drawn, for diagnosis.
4. Multiple breadth-first walks hand out integer lattice indices, hopping from cell to cell using *that cell's own* measured size times the known pitch-to-block ratio. Each connected hypothesis is fitted provisionally, and other blobs close to its integer grid sites are then recovered, which bridges a missed local hop without inventing an occluded cell.
5. A homography is fitted from integer indices to cell centres, every cell is re-scored against the footprint the fit predicts **for it**, and the fit is repeated on the survivors.
6. Every strongly supported mode-sized window (7 x 6 vertical, 3 x 10 horizontal) is retained, swept across both axes, ordered by distance from the image's bottom-left corner and capped at sixteen. The operator selects one and saves.

Validation and constraints. Colour parity, measured aspect ratio, mean residual and maximum residual are **hard acceptance gates**, not status readouts: a fit that fails any of them is refused rather than reported with a warning. A window needs at least 95 % physical coverage with every row and column supported, so one underlit edge cell does not move the calibration but a clipped strip cannot pass. Partial cells, clipped by the edge of the paper or the edge of the frame, are excluded from the fit entirely and not down-weighted, because their centres and their sizes are both wrong. And a refusal always still draws what it found, in green for blobs that joined a lattice and red for those that did not, with the count and the stage in the corner: a blank window would make 'the sheet is out of shot', 'the

colours are wrong' and 'the code never ran' look identical, which is precisely how the first live attempt was reported.

Which axis is machine X is never inferred from the image. The explicit mode decides it: vertical maps 2.2 cm to X, horizontal maps 6.0 cm to X, and the detected short and long lattice axes follow that declared geometry whichever way round the sheet was photographed. The complete 7 x 6 or 3 x 10 count is then cross-checked against the machine grid before calibration, so a partial sheet never causes an orientation guess.

When the gantry itself hides part of the sheet, an evidence-assisted variant of the same route pools accepted observations across several manually confirmed frames. It will fill an **interior** hole from the fitted geometry, and it will never fill an outer boundary: saving requires at least two accepted frames, at least four previously-verified cells of overlap between them, at least 60 % of the map physically observed, all four corner regions present, every outer edge supported, a merged residual of at most 2 px mean and 6 px maximum, and a repeated-cell spread of at most 3 px proving the camera and the paper did not move.

4.3.4 From pixels to a cell

The saved map is deliberately small: four envelope corners plus the grid geometry, not a per-cell table. A consumer loads it, spaces the cells evenly between those corners, and can then answer both directions of the question: which cell is under this pixel, and which polygon on the image is this cell. The browser carries a TypeScript port of the same arithmetic for the local hover highlight, while the Python implementation stays authoritative for what is actually selected, because the browser is never trusted.

A saved map is only adopted by a consumer whose **projection** (lens profile, flip and rotation, correction on or off, framing region of interest) matches the one embedded in the map. A mismatch is refused with a sentence naming what changed, and a map saved with no projection at all is refused by everything: it writes successfully and is then silently ignored, which looks exactly like it worked. A map on disk also does not reach a running console by itself, because calibration normally happens in a separate process; there is an explicit reload action for that, and a map that is present but refused must surface its reason, since 'no calibration saved' and 'the camera moved' need opposite responses from an operator and silence is indistinguishable from both.

4.4 Main System Operation

4.4.1 The fourteen-phase build cycle

One **B col row level** is one complete pick-and-place cycle. Every phase is announced before it runs, and every phase that can fail bails out immediately rather than carrying on with an unknown position.

Table 15: The fourteen build phases. Phases 1 to 11 place the block; 12 to 14 park the rig.

#	phase id	action	What the machine does
1	raise_clear	move	Raise Z into the top switch, clear of everything already built.
2	home_feeder	move	Home X and Y to the feeder cell [0,0]. Its centre is home, so there is no move afterwards.
3	neutralise_claw	rotate	Return the claw to neutral before picking up. Normally a no-op, because phase 14 already did; it also corrects a manual jog.
4	open_claw	release	Open the jaws, while still clear above the block.
5	lower_to_ground	move	Lower Z into the bottom switch. This also re-zeroes Z.
6	grip	grip	Close the claw. The block is now held.
7	lift_block	move	Raise Z into the top switch: carry height.
8	move_to_target	move	Home, then move Y to the row and X to the column, with the X-rail skew compensation applied.
9	rotate_to_grid	rotate	Apply the active grid's placement rotation, still above the stack.
10	lower_to_level	move	Lower Z to the target block level.
11	release	release	Open the claw. The block is placed. This is the one phase announced twice, with a done at the instant the jaws open.
12	park_clear	park	Raise Z clear of the block just placed. This must happen before any X/Y move, or the claw would drag through the stack it just added to.
13	park_home	park	Return X and Y to the origin.
14	park_rotation	park	Return the claw to neutral, if this build turned it.

Parking is not decoration. It leaves the machine in exactly the state a build expects to start from, so the rig is never left hanging over a stack with an open claw and the next **B** finds its first two phases already satisfied. It is also the one part of the cycle whose failures are treated differently: the block is already down by phase 12, so a parking failure cannot turn a good placement into a failed one, but it does leave the rig somewhere unknown, so the command's terminal result is downgraded from **OK** to **HELD**. Placed but not parked is not a success.

4.4.2 The feed cycle

One **FEED** <id> is one complete dosing operation on the Uno, and it is a state machine rather than a sequence of delays, so that a stop command remains available throughout.

Table 16: The feeder state machine.

State	Belt	Sensor sampled	Leaves when
closing	stopped	-	500 ms have elapsed
opening_stage_1	stopped	-	500 ms (gate at 90 degrees)
opening_stage_2	stopped	-	500 ms (gate at 160 degrees)
waiting_for_exit	stopped	exit, every 100 ms	a block is detected, or 10 s timeout
moving_to_stage	running	stage, every 100 ms	a block is detected, or 15 s timeout
aligning	stopped	-	350 ms (the aligner nudges and returns)
verifying_stage	stopped	stage, read once after settling	block still present -> success; block gone -> the belt resumes
block_ready	stopped	-	terminal success

Three moments in that sequence are the whole reason the feeder exists:

- **Before the container opens**, the stage sensor is read. If it already sees a block, the request is refused outright with **stage_occupied** and nothing moves. The pickup point is a single-owner resource, and an in-flight transaction is never cancelled and replaced, because its first block may already have left the hopper.
- **The instant the exit sensor fires**, the container is shut again, before the belt starts, so a second block cannot follow the first out. This is what turns a gate into a doser.
- **After the aligner has moved**, the stage sensor is read a second time. Only if the block is still there is **@id OK state=block_ready result=staged** emitted. If it has gone, the belt resumes instead of the feeder reporting a success it cannot see.

This is **sensor-stopped staging, not a fixed belt-duration guess**, and the two observations also distinguish an empty or blocked hopper path from a block that left the hopper and never arrived. The four terminal failures are **stage_occupied**, **exit_timeout** (no block seen leaving within 10 s), **stage_timeout** (a released block did not reach the pickup sensor within 15 s) and **cancelled**, and each names a different physical problem for the operator.

4.4.3 The guard stack

Every production placement passes through five layers, outermost first. Each one can refuse, and none of them can be skipped by the browser.

1. **The HTTP route** refuses a request that is not explicitly confirmed, one whose command string does not match the server's own computed command, a stale camera frame, a build that is already running, a locked session, or either board not being connected.
2. **The build worker** refuses a second build on the worker thread. One at a time, and off the event loop, so a blocking multi-minute call cannot stall the web service.
3. **The safety state** refuses everything while the session is locked, or when no cell is selected. This layer holds the selection, the level and the mode, and converts any exception from below into a session lock.
4. **The two-board handoff** refuses when its own lock is set, or when another placement already owns the pickup point.
5. **The serial clients** refuse an overlapping transaction, a board that is not connected, or a board whose identity does not match the configuration.

Above all five sits the rule that the browser is a mirror. It sends a request and renders what the server reports; every guard is re-checked on the Pi, and a greyed-out button in the interface is a courtesy, never the safety mechanism.

4.4.4 The two-board handoff

One placement is **one indivisible operation owned by the Pi**: stage exactly one block on the Uno, then, only on its exact terminal success, place it with the Mega.

```

Pi (CellOrchestrator)          Uno (belt_v1)          MEGA (build_test_v1)
phase=feeding
feed(timeout=45s) ---- FEED <id> -->
    <-- @id RECV cmd=FEED
    <-- @id ACK cmd=FEED accepted=1
    <-- @id STATE state=closing ... verifying_stage
    <-- @id EVENT phase=... (progress only)
    <-- @id OK state=block_ready result=staged <- the ONLY success
phase=ready_for_pick
phase=placing
build(col,row,level) ----- B <col> <row> <level> -->
    <-- @seq RECV cmd=B ...
    <-- @seq STEP step=n/14 phase=... status=begin
    <-- @seq OK col=... row=... level=... <- PLACED
phase=complete -> BuildResult(PLACED)

```

The rules enforced on that sequence are absolute:

- **Only** a terminal **@id OK state=block_ready result=staged** whose **id** matches the request authorises the **B**. **ACK**, **STATE**, **SENSOR** and **EVENT** are progress and never success. A terminal with the wrong id is ignored.
- A malformed success, meaning anything other than **result=staged**, is treated as a **failure**, not as permission.
- On any feeder error, timeout, disconnect or reset before the **OK**: **no B is sent**, the orchestrator locks and the result is **ABORTED**.
- After the **B** has been sent, **any** non-placed Mega result also locks, including a **SAFE** rejection, which is entirely safe for a bare gantry but not for the cell, because a block is already staged and feeding another would double-load the pickup point.
- One **FEED** at a time. The next one never starts until the previous **B** has returned a terminal **PLACED**.

Table 17: Result and lock behaviour.

Outcome	Result	Session	Recovery
Uno staged, Mega OK	placed	READY, selection cleared	continue
Uno error / timeout / reset before B	aborted	LOCKED	inspect the pickup area, restart the service
Uno OK , then Mega SAFE / rejected	aborted	LOCKED	a block is staged; inspect, then restart
Uno OK , then Mega HELD / timeout / cable loss	aborted	LOCKED	machine state unknown; inspect, then restart
Operator stop during the feed	aborted (cancelled)	LOCKED	inspect, then restart

The lock is deliberately sticky: a new service process is the required recovery. Nothing is auto-retried, because a retried **FEED** or **B** risks a double-load or a duplicate placement, and neither of those announces itself.

Stop has two meanings depending on where the operation is. While the Uno owns it (feeding or staging), stop actively sends the Uno a **STOP**, the feed waiter observes a

terminal **cancelled**, and the session locks for inspection. Once the Mega is moving, stop is **stop after the current block**, because the Mega does not read serial inside its build cycle. The interface says exactly that, in those words, and never offers a control that implies otherwise.

4.5 Additional Operating Modes

4.5.1 Browser click-to-build

What it does: Lets an operator place one block by tapping a cell on the live camera image.

Inputs: A tap on the camera view, a level from a stepper control, and two confirmations.

Outputs: One complete guarded feed-then-place operation, and a terminal result.

Typical workflow: 1. Tap a cell. This selects it; nothing moves. 2. The panel shows the exact command, **B 3 2 1**. 3. Tap BUILD, then CONFIRM. 4. The server sends one correlated **FEED**, then the **B**. 5. The result is **PLACED** (green, selection cleared), **REJECTED** (amber, nothing moved, selection kept) or **ABORTED** (red, session locked).

Validation and constraints: BUILD is disabled unless both boards are connected. Every mutation is refused while a build is running. A stale camera frame blocks selection and build. There is no cancel and no retry control, because the machine cannot honour either.

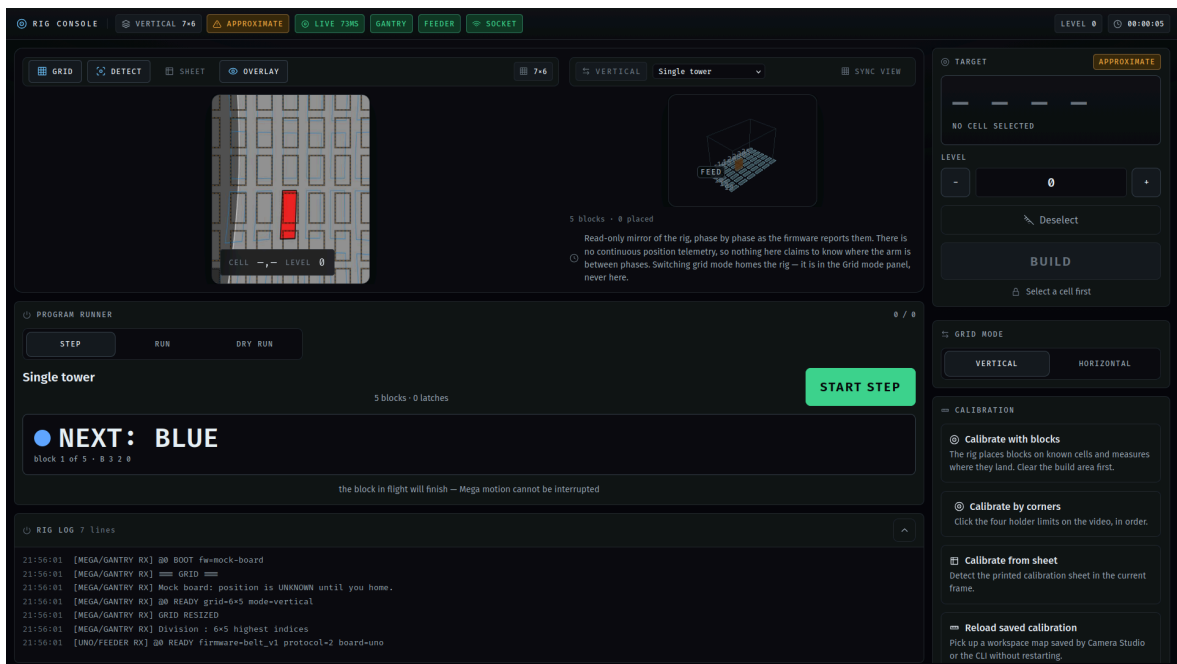


Figure 15: The browser operator console: live camera view with the grid overlay, the selected cell and its exact command, the level stepper, and the status rail.

4.5.2 The 3D Build Studio

The Studio is the flagship of the software side and the normal entry point for building a structure. It is a design environment, a physics validator, a compiler and an execution runner, all against the machine's real lattice.

What it does: Lets a person design a 3D structure block by block on the machine's own grid, tells them immediately whether it is physically buildable, compiles it to an ordered command program, and then runs that program against the rig one guarded block at a time.

Inputs: Mouse or touch placement in a live 3D scene; the machine's grid geometry, imported directly from the same configuration file the Pi reads, so there is no second copy of the geometry anywhere.

Outputs: A saved **rigmodel/1** design in the browser's own library; a compiled program of **B col row level** commands separated by **R** and **RR** latches; and, after a run, a Markdown run report.

The validator answers three physical questions before anything is compiled:

- **Support:** does the block rest on something, and is that support a centre-of-mass toppling test rather than a contact-ratio shortcut?
- **Collision:** does it overlap a block already placed, in machine space and not in grid indices, so that a horizontal block spanning two vertical columns is checked correctly?
- **Reachability:** is the cell inside the active grid, and is the level inside the firmware's build ceiling?

The compiler is the intellectual core. **B col row level** carries no orientation, so a mixed-orientation model is a partial order that has to be sorted into same-mode runs, each run costing a homing move for its latch. It runs in four named steps:

1. **Support graph:** block id to the ids it rests on, from machine-space footprint overlap at the matching base and top heights. A horizontal span depends on every vertical stack under it.
2. **Ordering:** Kahn's algorithm, with the ready set re-sorted on every pop by a comparator chain: by level (bottom-up, so a block can never precede its support), then by the currently latched mode (stateful, recomputed against the mode the emitter would be in by then), then by the author's own order wherever that is still legal, then by cell, then by id so that ties cannot exist.
3. **Emission:** a one-variable latch state machine that emits a mode command only on an actual change, annotated with its cost.
4. **Summary:** blocks, latches, levels and a time estimate.

Two properties of the compiler are enforced by tests and not asserted in prose. **It is deterministic**: the same model compiles twenty times to byte-identical output, and the program depends only on the author's order and not on the internal array order. And **an invalid model compiles to nothing at all** (an empty program, zeroed statistics and the diagnostics) rather than to a half-program, because a half-program is the single most dangerous artefact this codebase could produce.

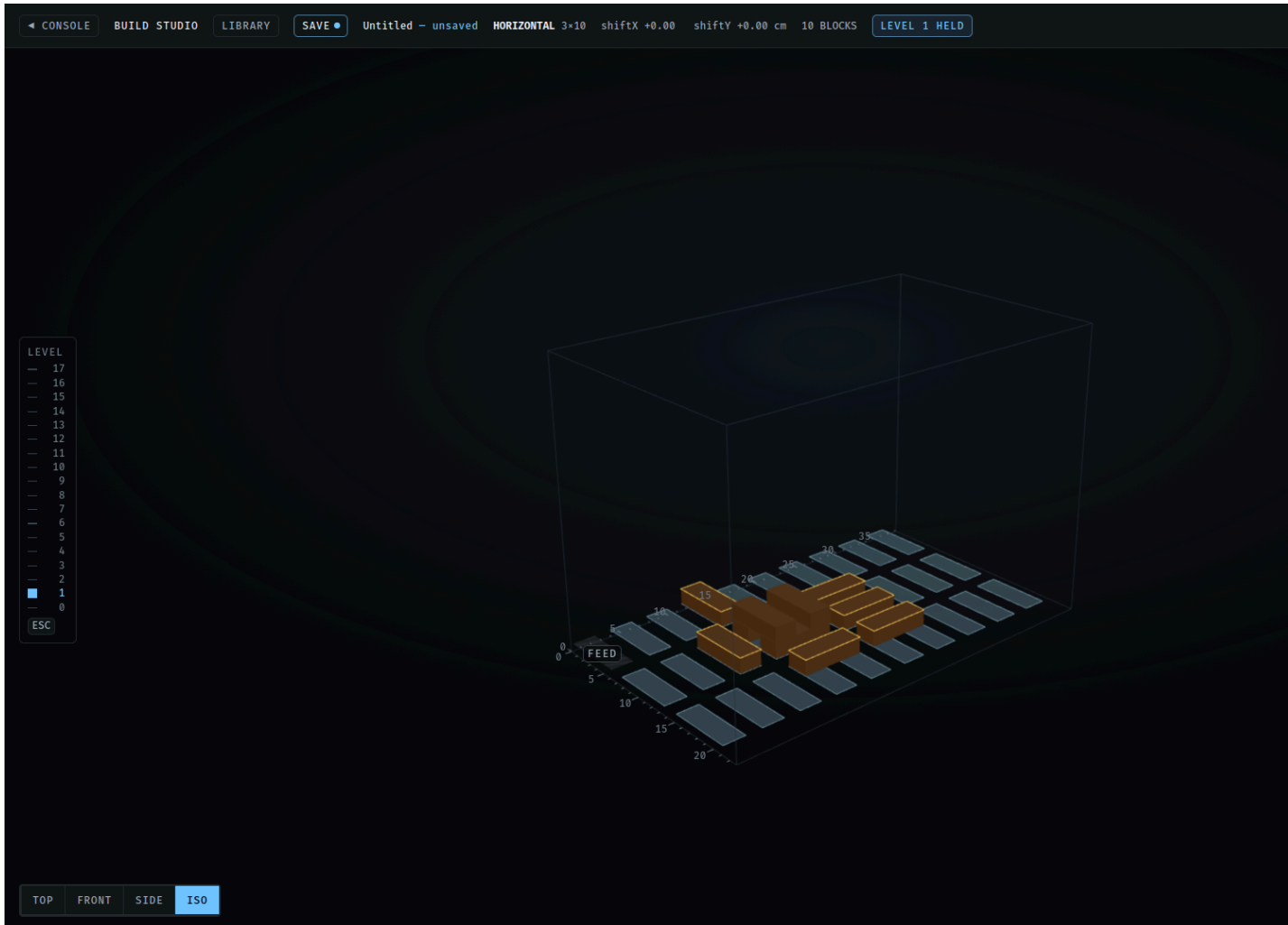


Figure 16: The 3D Build Studio: the design scene on the machine's lattice, the level scrubber, the diagnostics panel and the compiled program view.

4.5.3 The live digital twin

Beside the camera view sits a 3D twin of the rig that mirrors the real build as it happens. The whole of it is one pure function from the server's state to a picture, and the component simply draws what that function returns, which is what makes the twin's behaviour testable against recorded server sessions and not only observable.

Two rules define it:

- **It never invents state.** A block is drawn as placed because the server reported a terminal placed result for it, and for no other reason. There is no optimistic placement.
- **After an abort it stops.** A locked session freezes the animation, desaturates every block, drops the target marker and demotes everything unconfirmed to a ghost. The machine's real state is unknown after an abort, and a twin still cheerfully rendering the plan is at its most misleading exactly when misleading is most expensive.

The twin is **phase-driven**, from the firmware's own **STEP** lines, and it deliberately carries no descent timer of its own. An earlier version interpolated a looping 1.6-second descent from the browser's clock and returned a height nobody had measured; it looped, so a forty-second build showed twenty-five descents that had never happened. What replaced it says only what the firmware said. The carried block has exactly two heights, travel and zero, and it drops to zero on the release event, which is a fact. The one animation a clock drives is the placement descent, and it is fenced four ways: it starts from the moment the phase event arrived, its duration is the machine's own **ms=** arithmetic, it is clamped short of the cell so it **cannot** reach it, and with no **ms=** it does nothing at all rather than guessing. If Z jams, the block glides down, stops just short and sits there visibly not landing, which is the truth.

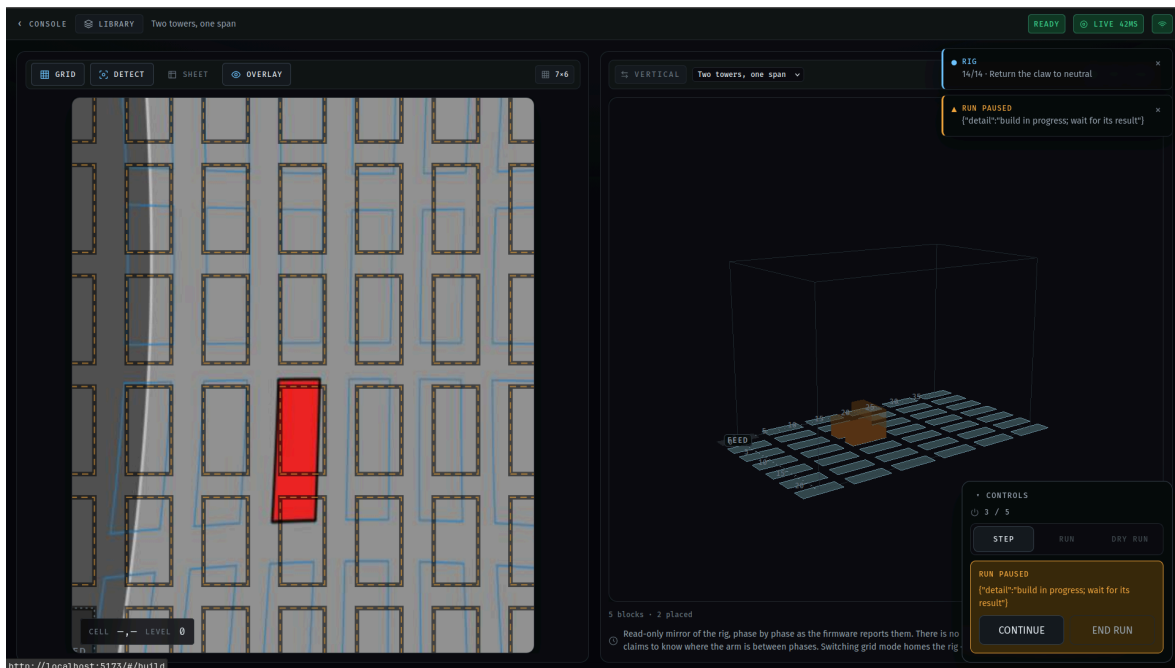


Figure 17: The live digital twin beside the camera view during a build, showing the carried block, the current phase and the confirmed placements.

4.5.4 The execution runner

The runner takes a compiled program and dispatches **one guarded request at a time**, waiting for the durable terminal result before the next. It offers STEP (one block), RUN (continuous) and DRY RUN (the whole program with no API traffic at all, for rehearsal),

shows the current command, the blocks placed against the total, the elapsed time and an estimate from measured cycle time, and exports a Markdown run report of the session.

It never relaxes a safety rule. A **HELD** result locks the run; a **SAFE** rejection pauses it and keeps it resumable; a lost socket pauses and resumes on the next phase event and not on a timer; and 'STOP AFTER THIS BLOCK' becomes 'CANCEL FEED' while the feeder still owns the operation, because during that window a stop genuinely can act. The panel's next-block colour line is a preview only, and never implies that a second feed is already running.

4.5.5 Commissioning and simulation modes

Full simulation: The entire software chain (web service, orchestrator, both serial clients and camera) runs against a protocol-level fake Mega, a protocol-2 fake Uno with failure, reset and cancel controls, and a mock camera that renders blocks at real grid cells. No hardware at all. This is how the console and the Studio are developed and rehearsed, and how the runner's cycle-time constant was measured.

Desktop click-to-build: The original OpenCV tool the browser console replaced. It still works, and it is the fastest way to drive the rig from the machine it is plugged into: click a cell on the camera view, choose a level, confirm the displayed command.

Serial commissioning consoles: One per board. A direct line to the Mega for homing, jogging, grid reports and single builds, and one to the Uno for status, feed, belt direction and the container gate. These are the tools the commissioning checklists are written against.

Standalone calibration tools: Camera Studio, which owns the lens, colour, sensor and framing settings and writes them to disk; the printed-sheet detector on its own; the evidence-assisted collector; and the placed-block calibrator with a hardware-free dry-run mode.

The mock mode deserves one more sentence, because it is the reason the software could be developed at all. The rig is a shared, slow, physically hazardous resource that takes half a minute to answer a question. A protocol-level fake that speaks exactly the same acknowledgement grammar turns that half a minute into a millisecond, and it is why the two-board handoff, the session-lock behaviour and the whole browser stack could be tested exhaustively without ever risking the machine.

5. Testing, Results and Discussion

5.1 Testing Methodology

5.1.1 Four levels, in order

The rig is a slow, shared and physically hazardous test fixture: it takes half a minute to answer one question, and a wrong answer can drive a claw into a stack. The testing strategy is built around that, and it moves work down to the cheapest level that can still answer the question honestly.

1. **Automated software tests, no hardware.** Every pure function (the grid arithmetic, the serial protocol parsers, the compiler, the twin's state mapping, the runner's state machine, the workspace homography) is tested off the machine. The two grid implementations, one in Python and one in TypeScript, are held to each other by fixtures dumped from the Python side, so the browser cannot silently disagree with the Pi about where a cell is.
2. **Protocol-level simulation.** A fake Mega and a fake Uno speak exactly the same acknowledgement grammar as the real firmwares, including their failure, reset and cancellation behaviours, and a mock camera renders blocks at real grid cells. The complete browser-to-orchestrator-to-serial path runs against them, which is how the two-board handoff and the session-lock behaviour were tested exhaustively without ever risking the machine.
3. **Firmware host builds.** There is no Arduino toolchain on the development machine, so firmware changes are compiled against a stub-Arduino host harness that proves the sketch parses and that its acknowledgement output is byte-for-byte what the protocol specifies. A clean compile is explicitly **not** treated as a test of behaviour.
4. **Hardware bring-up and runs on the rig.** Anything touching motion, limits or Z is flashed to the real board and watched. Every real run appends to two log files: one stopwatch per build, and every line to and from either Arduino with the gap since the previous line. Those logs are the primary evidence in this chapter.

The conditions for the hardware runs were the ordinary working conditions of the lab: the rig on its table under the room's normal overhead lighting, the camera fixed on its support structure, the surface clear at the start of each session, and the operator present throughout. Vision tests were run both live and against saved reference captures, so that a detection result can be reproduced exactly instead of depending on the light on the day.

5.1.2 Pass and fail criteria

Motion: Every axis finds its home switch within the homing cap; a commanded cell is reached without a soft-limit refusal; a repeated command produces the same phase timings; and no cumulative positional drift is observable across a session.

Manipulation: The claw picks up a single staged block, holds it through the whole carry, releases it on the commanded cell, and the rotation returns the claw to neutral.

Feeder: Exactly one block reaches the pickup point per request; the exit and stage sensors both fire in the right order; and each of the four terminal failures can be provoked deliberately and reports the right reason with the belt stopped.

Vision: Every block on a full board is found, with no false detections from the rails or the holder offcuts, and the fitted grid passes its parity, aspect and residual gates instead of merely reporting them.

Protocol: Exactly one terminal acknowledgement per command; **SAFE** and **HELD** never confused; and a non-terminal line never satisfies a waiter.

Integrated: A complete structure is built from a compiled program with no manual intervention, and any failure locks the session instead of continuing.

5.2 Motion System Testing

The results in this section come from **sixteen complete B commands executed on the physical rig on 3 September 2026**, across five separate service sessions, all with the hardware link live and not mocked. Every one of the sixteen returned a terminal **PLACED**. The gantry firmware's own **@seq STEP** lines were captured with wall-clock timestamps, so each of the fourteen phases has an independently measured duration.

5.2.1 Cycle repeatability

The clearest motion result in the project is how little the fixed part of the cycle varies. Nine of the fourteen phases have no distance or level dependence at all, and across all sixteen builds those nine phases summed to a **mean of 12.57 s with a standard deviation of 0.18 s**, a spread of 1.4 %. Six of the nine repeated with identical minimum and maximum to the logging resolution of 0.01 s.

Table 18: Per-phase durations over all sixteen hardware builds, in seconds.

#	phase	Min	Max	Mean	Comment
1	raise_clear	0.42	0.43	0.42	Z seek to the top switch
2	home_feeder	0.57	0.58	0.58	X/Y home; already there after a park
3	neutralise_claw	0.41	0.41	0.41	no-op after a parked build

#	phase	Min	Max	Mean	Comment
4	open_claw	0.93	0.93	0.93	600 ms servo settle + 250 ms phase pause
5	lower_to_ground	2.91	2.91	2.91	full Z travel down, seek to the switch
6	grip	0.95	0.96	0.95	600 ms servo settle + 250 ms phase pause
7	lift_block	2.89	2.89	2.89	full Z travel up, loaded
8	move_to_target	1.48	7.12	4.53	distance-dependent
9	rotate_to_grid	0.39	1.92	0.49	0.39 s no-op; 1.92 s for a real 90 deg turn
10	lower_to_level	2.26	2.92	2.80	level-dependent
11	release	0.70	0.72	0.71	600 ms servo settle, no phase pause
12	park_clear	2.25	2.86	2.76	Z lift from the placement level
13	park_home	1.40	8.53	4.68	distance-dependent
14	park_rotation	0.40	1.96	0.52	as phase 9

That repeatability is a direct consequence of the design choice in Section 3.1.2. With no acceleration ramp, a move of N steps takes N times a fixed period, and there is nothing left to vary. It is also what makes the variable phases interpretable: when phases 8 and 13 range from 1.48 s to 8.53 s while everything around them does not move at all, the variation is travel distance and nothing else.

Three of the fixed phases can be predicted from first principles, and all three agree:

- **Phases 4 and 6 (servo open and close)** should take the 600 ms commanded settle plus the 250 ms inter-phase pause, which is 0.85 s. Measured: 0.93 s and 0.95 s.
- **Phase 11 (release)** has the same 600 ms settle but is measured to the **done** line and not to the next phase, so it should be about 0.6 s. Measured: 0.71 s.
- **Phases 9 and 14 with a real rotation.** A 90-degree turn is 512 of the 28BYJ-48's 2,048 steps at 10 rpm, which is 1.50 s. The one horizontal-mode build in the log spent 1.92 s on phase 9 against 0.39 s for the no-op case, a difference of **1.53 s**.

The residual 80 to 110 ms on each of those is the logging and serial round-trip overhead, and it is consistent across every phase, which is what makes it believable as an overhead and not as an error.

5.2.2 Homing repeatability and lost steps

Homing repeatability was not measured with an instrument, but the sixteen builds carry strong indirect evidence. Phase 2 (home X/Y to the feeder) took between 0.57 and 0.58 s on **every single build across all five sessions**, and phase 1 (Z seek to the top switch) took between 0.42 and 0.43 s on every build after the first of each session. A homing move whose duration is constant to 0.01 s is a homing move that starts from the same place and finds its switch after the same travel every time. If the machine were losing steps, the parked position would drift and the homing travel, and with it the homing duration, would grow.

No lost-step event was observed in any of the logged sessions: every build returned **PLACED**, no soft-limit refusal was recorded, and the statistics the firmware keeps for short moves and blocked moves stayed at zero. That is not the same as proving there were none, and the honest statement is that the re-home before every move makes a lost step self-correcting and not detectable.

5.3 Manipulation Testing

5.3.1 Grip and carry

Across all sixteen logged builds and the commissioning runs that preceded them, **no block was dropped, crushed or lost from the claw during a carry**. Every build reached its phase-11 release and its terminal **OK**. The grip is a single fixed closed angle of 52 degrees on the jaw servo, tuned on the rig until the jaws held a block firmly without marking it, and it is not adjusted per block or per level.

The failure mode the design guards against is not a weak grip but a **mistimed** one: the servo reports nothing, so a build that started moving Z while the jaws were still swinging would either miss the block or crush it against the surface. The fixed 600 ms settle after every open and close is the answer, and Section 5.2.1 shows that it is honoured exactly, at 0.93 s and 0.95 s including the phase pause.

5.3.2 Rotation

The 28BYJ-48 rotation repeated accurately with no position sensor and no observable drift over a session. The single horizontal-mode build in the log gives the one direct timing measurement: 1.53 s of net rotation time against the 1.50 s that 512 steps at 10 rpm predicts, a 2 % agreement, and the build completed with a terminal **OK** and a clean return to neutral at phase 14.

The design leans on the fact that a tracked angle is never more than one cycle old, since the claw is returned to neutral over the feeder before every pickup and again after every placement, rather than on the stepper being perfect. The clearest stacking result in the log

is the afternoon session of 3 September, in which cell [3,2] was built up **level by level from 0 to 4**, five blocks on one cell, one command per level, all five returning **PLACED**:

Table 19: The five-level stack on cell [3,2], from logs/build.log.

Time	Command	Level	Phase 10 descent (s)	Total (s)	Result
13:51:07	B 3 2 0	0	2.91	31.42	PLACED
13:51:39	B 3 2 1	1	2.70	32.19	PLACED
13:52:11	B 3 2 2	2	2.56	30.76	PLACED
13:52:42	B 3 2 3	3	2.41	31.64	PLACED
13:53:14	B 3 2 4	4	2.26	29.84	PLACED

The descent shortens monotonically by a mean of 0.160 s per level, which is the machine measuring its own block height: 1.5 cm at 50.94 steps/cm at 1.9 ms per step is 145 ms per level. That is a satisfying result, because it means the stack really is growing by one block height per level as far as the Z axis is concerned, and not by whatever the blocks happen to settle to.

The fixed Z margin of +0.10 cm at levels of 1 and above was introduced **after** this session, so these five placements ran with no margin at all and still completed. The firmware's own ceiling is level 16 at 24.0 cm, leaving 2.5 cm of headroom to fly a block over the tallest possible tower.

5.4 Vision System Testing

5.4.1 Block detection

Detection was measured against two saved reference boards, both of which are the vertical grid with **29 blocks** laid from the home corner (columns 0 to 6 of rows 0 to 3, plus a single block on [0,4]), leaving 13 cells unplaced. Both boards also carry the holder's two thin wooden offcuts beside cell [0,0], which are the most useful thing in them: they are wooden, roughly block-shaped, and not blocks.



Figure 17: The vision reference board: 29 blocks on the vertical grid, the aluminium rails at the frame edges, and the holder's two wooden offcuts beside cell [0,0]. The magenta cast is the camera's uncorrected colour response.

On both boards the detector found **29 of 29 blocks**. The raw segmentation proposes 33 hypotheses; the duplicate, frame-edge and lattice rejection steps take that to 29, and the four it removes are the aluminium rails and the two holder offcuts. The lattice snap error is at most **0.08 cells** and the mean fit residual **0.85 px**, every outline comes out square with one shared size and one shared bearing, and the whole pass costs 37 to 103 ms at the shipped settings.

Two negative results are worth as much as the positive one, because both closed off an obvious 'improvement'. **Higher resolution and illumination flattening find nothing extra and cost up to four seconds a frame:**

Table 20: Detection count and cost against processing width and illumination flattening.

Processing width	Flatten	Blocks found	Reference board	Full 1296 px frame
384 px	off	29 / 29	84 ms	103 ms
384 px	on	29 / 29	291 ms	221 ms
480 px	off	29 / 29	37 ms	104 ms

Processing width	Flatten	Blocks found	Reference board	Full 1296 px frame
640 px	on	29 / 29	578 ms	902 ms
1024 px	off	29 / 29	60 ms	574 ms
1024 px	on	29 / 29	543 ms	3,894 ms

The 33-to-29 improvement is entirely the rejection steps, not the segmentation. Lowering the minimum contour area from 500 to 250 px likewise finds nothing extra and doubles the cost, because every small contour it admits enters the rectangle search. A timing guard in the test suite keeps this conclusion from being quietly reverted.

The second negative result is about white balance, and it is deliberately backwards from what the sheet detector does. A white-patch balance is safe on a frame that is mostly white paper, and unsafe on a board covered in wooden blocks: the bright quantile lands partly on the wood and the correction then pulls the blocks toward the surface it was meant to separate them from. Measured on the reference board, **balance on finds 28 of 29 and balance off finds all 29**, and off holds at 29 across every colour threshold from 4 to 8 where on collapses at 4.

5.4.2 Calibration results

The best-evidenced calibration figures on this machine come from fitting the grid to blocks the rig itself placed, because the thing measured is then the thing being calibrated: the fit absorbs backlash, the tool offsets and the real placement error rather than a printed approximation of them. Those results are reported here because they are the numbers that actually exist; the printed-sheet route of Section 4.3.3 writes a byte-identical artefact and is held to the paper route field by field by an automated parity test.

Cell pitch: X measures 29.38 px with a standard deviation of 1.24, a 1.4 % spread across rows; Y measures 69.14 px, sd 1.08, a 2.0 % spread across columns. The pitch is measured from lattice-adjacent pairs only, so every sample is exactly one pitch, and it is reported per row and per column as well as pooled, because 'is the gap a static number' is the question a single average hides. Static to within 2 %: on this rig the answer is one number per axis.

Model selection: Four models compete on leave-one-out prediction, not on training error, because the point of the fit is to place cells no block was ever put on. Held-out error came out at 101.4 px for a similarity, 1.38 px affine, 1.22 px homography and **1.09 px for homography plus curvature**, which is the model chosen. Its fit residual is 0.85 px mean, under 3 % of a 29.38 px cell pitch.

Anisotropy agreement: The one non-circular geometry check: the optical stretch measured from cell **pitches** against the stretch measured from block **footprints**. The reference board's view is genuinely 17.7 % anisotropic and the two independent

estimates agree to 4 %, which they only can if the configured gaps really do describe this board.

Saved-map flattening error: 1.25 px mean and 2.07 px maximum, which is 0.27 cm on a 2.2 cm block. The saved artefact carries four envelope corners and not a per-cell table, so a consumer spaces cells evenly between them and the curvature term is flattened on the way out. The corners come back exact and the error peaks mid-grid, which is that flattening's signature.

Conditioning gate: Degenerate configurations score $1e-17$ and below, usable ones above $1e-2$, and the gate sits at $1e-5$, in a thirty-order-of-magnitude gap. Spread and hull area are necessary and insufficient: after seven row-major placements the set is six points along one row plus one, which is completely degenerate, so the gate is a numerical conditioning number and not a cell count.

Two limits on those numbers have to be stated. First, **they are measured on saved reference captures rather than on a live camera frame each time**, which is what makes them exactly reproducible in a test suite and also what stops them being a claim about the light in the room on any given day. Second, the 0.27 cm flattening error is a property of the saved file format and not of the fit: removing it means widening **workspace_map.json** to carry a per-cell table, which touches every consumer of the format.

One failure is worth recording because refusing was the correct behaviour. A live frame of the printed sheet found 89 coloured blobs with 53 of them on the lattice, spread over 11×6 , but with holes, from a cable lying diagonally across the sheet and from the frame clipping the outer columns, so the largest unbroken run was only 6×3 . The strict single-frame route refused it, and it was right to: that photograph genuinely does not contain six whole rows. Refusing a fit that cannot be supported is the behaviour the gates exist for, and it is what the evidence-assisted multi-frame variant was built to work around.

5.4.3 Colour correction

The colour transform was fitted by matching a live camera frame against a phone photograph of the same scene, iterating on a similarity metric:

The saved profile records the outcome. Similarity rose from a baseline of **0.404** to **0.669** over six iterations, against a target of 0.95 that was **not met**, and the result is deliberately applied at 76 % strength. Even softened that far, the darkest 2 % of the frame falls from 122 to 74, and no new clipping is introduced.

Separately from that transform, the sheet detector's own internal white-patch balance was measured against the rig's real ink colours, and the result is the clearest single number in the vision work: **without balancing, the green mask contains 0 pixels; with balancing it contains 87,836**. Not a degraded detection, but no detection at all. With balancing on, the same frame fits a 7×6 grid from 42 of 42 window cells at 0.28 px mean residual and 100

% colour parity, and the fit survives a magenta cast, a blue cast and a warm cast without changing a threshold.

The phone-matched transform stops well short of its target and the reason is recorded in the profile itself: the fitted matrix wants an offset of 165 levels, which is far outside the brightness range the scene could actually measure, and applying it at full strength crushes the shadows (the darkest 2 % of the frame falls from 122 to 74 even at 76 % strength). This is a case where the software correction has reached its useful limit and the remaining error belongs to the optics: a lens with this much veiling glare cannot be white-balanced into a clean image, and the correct fix is the camera and the lens rather than another iteration of the matrix.

5.5 Sensor Testing

5.5.1 Ultrasonic sensors

Both HC-SR04 sensors were commissioned against the documented checklist: with the sensors clear, a status snapshot must report a real distance or **no_echo** and not a false nearby block; a block placed at each sensor in turn must cross the 10 cm threshold only at that sensor; and each of the four terminal failures must be provokable on purpose: start with the stage occupied, leave the hopper empty, obstruct the belt, and cancel a running feed, with the right reason reported and the belt stopped in every case.

The 10 cm threshold proved reliable at both positions and did not need re-tuning from its installed value. The design detail that made it reliable is a small one: a sensor timeout returns -1 and is reported on the wire as **distance_cm=no_echo**, and that is **never** treated as a detection. 'I heard nothing' and 'nothing is there' are different statements, and a feeder that collapsed them would decide an empty belt was fine.

5.5.2 Limit switches

All four switches performed without a missed trip or a false trigger in any logged session; the machine's own boot report reads the state of each one and every session in the log opened with a correct reading (the X and Z-top switches showing as hit, because the rig had been left parked at the origin with Z at the top, which is exactly where the previous build's park phase leaves it).

The 200 us confirm time was sufficient. Contact bounce on a micro switch is typically in the range of a few hundred microseconds to a few milliseconds, so 200 us on its own would not be a complete debounce, but it does not have to be, because the switch is polled once per 1.15 ms step and a bounce that briefly reopens the contact simply means the axis takes one more step before the trip is confirmed.

5.6 Challenges and Limitations

5.6.1 The problems that changed the design

The slanted X rail: Placements involving X travel landed off along Y by an amount proportional to the column index. Diagnosing it as a rail out of square rather than as a calibration error took a measurement per column: a constant offset and a proportional one look identical at a single cell. **Solved** by a firmware Y nudge applied only to the build motion (Section 4.2.5), flashed and re-tuned on the rig from 0.15 to 0.10 cm per column. The residual has not been re-measured.

A correction applied to the wrong quantity: Horizontal placements landed 1.4 cm too far from the X home switch. The grid's per-mode error offset was reached for first, and it absorbed the symptom while hiding a sign error: the error offset is one rotation-blind number per mode, so it could not follow when the build's rotation settled on clockwise, and its X half then pushed the **same** way as the real error instead of against it. **Solved** by identifying the actual cause, which is that the grip point sits about (-0.3, +0.6) cm off the rotation axis, so a 90-degree turn swings the block centre round it, and moving the correction into the per-rotation tool offset where it belongs. This is the clearest lesson of the project: **a correction applied to the wrong quantity can fit the data and still be wrong**, and the only defence is keeping the offset families separate enough that they can be told apart by measurement.

A camera cast that broke detection outright: The first live frame of the printed calibration sheet detected nothing at all, and the failure was invisible from the outside: the green ink had moved to hue 120, which is cyan, missing the hue window and the saturation floor at once, while the pink wall behind the rig landed inside the magenta window. **Solved** in two layers, a white-patch normalisation inside the sheet detector and a saved colour transform applied once at the captured frame. **Partly unsolved:** the fitted transform reaches only 0.669 similarity against a 0.95 target, and the remainder belongs to the optics.

A refusal that looked like a crash: The same first attempt was reported as 'no detection, no overlay, nothing', which made 'the sheet is out of shot', 'the colours are wrong' and 'the code never ran' indistinguishable. **Solved** by making every refusal still draw what it found, in green for blobs that joined a lattice and red for those that did not, with the count and the stage named. A diagnostic that goes blank on failure is worse than no diagnostic.

A calibration that saved successfully and did nothing: The workspace map is read once at startup, and calibration normally runs in a separate process, so a freshly saved calibration was invisible until the service was restarted and nothing said so. Saving looked like it worked. **Solved** with an explicit reload action on every consumer, and with the rule that a map which is present but refused must surface its reason: 'no

calibration saved' and 'the camera moved' need opposite responses from an operator, and silence is indistinguishable from both.

Forty seconds of silence: The gantry does not read serial during a build, so the Pi learned nothing between sending a command and its result. Every interface that wanted to show progress had to invent it, and one early version of the twin did exactly that, looping a 1.6-second descent animation twenty-five times during a single forty-second build. **Solved** by having the firmware narrate its own phases on a machine-readable channel, and by deleting every browser-side timer that was not driven by something the machine actually said.

A board that forgets everything: Opening the USB port resets the Arduino, which comes back un-homed and in its compiled default mode with no memory of the grid size or any shift. **Solved** by re-pushing the mode latch, then the grid size, then any shift on every connection, in that order (the order matters, because the size is validated against the active mode), and by treating an unexpected boot line mid-session as a hard error.

5.6.2 Limitations of the delivered system

These are stated without softening, because a report that implied otherwise would be describing a machine that does not exist.

- **No hardwired emergency stop, and no interruptible motion.** There is no button, no safety relay and no enable chain that removes motion energy independently of the software, and the gantry cannot be stopped from software once it is moving. The stop of last resort is removing power at the supply. The machine is run attended.
- **No watchdog.** If the Pi crashes, the browser closes or the USB cable is pulled mid-motion, the Mega finishes the command it already holds. Nothing tells it the master is gone.
- **Open-loop steppers with no stall detection.** No encoders, no current sensing. A mechanical stall produces step pulses that go nowhere and the machine does not know.
- **The lens is tuned, not calibrated.** The parameters were adjusted by eye against straight edges. The image is visually straight and is not measurement-grade, and the saved calibration file itself records **source: estimated**.
- **The claw angle is not sensed.** The operator is trusted to start with the claw neutral. If it is not, every placement in that session is turned 90 degrees and nothing in the system can tell.
- **No OS-level exclusive lock on either serial port.** One process is meant to own each board, and that is an operational rule rather than something the operating system enforces; running a commissioning script while the service is up will contend for the same board and reset it.

- **The saved calibration format flattens the fit.** Four envelope corners cannot carry the curvature term, which costs 0.27 cm of mid-grid error on a 2.2 cm block.

6. Conclusion and Future Work

6.1 Conclusion

This project set out to build a machine that constructs a human-designed 3D block structure with no human placing a block by hand, and that confirms every physical transition with a sensor or a camera rather than with a timer. The result is a complete robotic cell: a CoreXY gantry with an added Z axis and a rotating mechanical claw, a separate feeder that doses one block at a time and proves it twice, an overhead vision system that maps camera pixels to grid cells, and a browser software chain in which a person designs a structure, compiles it, and watches a live digital twin mirror the real build from the machine's own telemetry.

The machine works. Sixteen complete pick-and-place cycles were executed on hardware and all sixteen completed, at 19.8 to 32.2 seconds per block, with the fixed part of the cycle repeating to within 0.18 s across every one of them. A five-level stack was built one command at a time on a single cell, and the descent shortened by 0.160 s per level against the 0.145 s the block height predicts, which is the machine measuring its own geometry. Both grid orientations ran on hardware. The vision system finds all 29 blocks on a full board while correctly rejecting the aluminium rails and the two wooden offcuts that look exactly like blocks, and the grid fit selects its own model on held-out prediction error and not on training error. The software is covered by 492 browser tests and 68 Python protocol tests.

Measured against the objectives of Section 1.3:

1. **Reach and place on every cell, at any level. Met.** 41 buildable vertical cells and 29 horizontal, at levels 0 to 16 by the firmware ceiling. Levels 0 to 4 and columns 0 to 5 were exercised on hardware.
2. **Two block orientations. Met.** Two separately calibrated grids, latched by a command that moves nothing; both were exercised on hardware.
3. **A feeder with confirmation at both ends. Met in design and in firmware,** with both sensors, all four terminal failures and the two-stage gate implemented and commissioned. The full feed-to-place chain on hardware is the one piece of evidence this report could not produce from the logs (Section 5.6.2).
4. **Vision detection and pixel-to-cell mapping. Met.** 29 of 29 detection with correct rejection of non-blocks, a 0.85 px mean fit residual, and 0.27 cm of mapping error through the saved format.
5. **Browser operation. Met.** Live camera view, tap to select, two-tap confirm, from a phone on the local network.

6. **Design, validate and compile a structure. Met.** The Studio validates support, collision and toppling, and compiles deterministically to an ordered program with minimal mode latches.
7. **The safety model. Met in software,** at five independent layers, with **SAFE** and **HELD** as separate protocol kinds so that the dangerous retry cannot be written by accident. **Not met in hardware:** there is no emergency stop, and that gap is stated throughout and not hidden.
8. **Cycle time under 45 s. Met.** 26.1 s mean and 32.2 s worst case, for a direct gantry build.

What the project demonstrates best is not any one of those. It is the discipline underneath them: a design in which no part of the system is allowed to assume what another part has actually done. The feeder does not report success because time passed; it reports success because a sensor saw a block and then saw it again after the aligner moved. The gantry does not tell the Pi it is fine; it narrates each of its fourteen phases before running it and predicts how long the slow ones will take. The twin does not animate a placement; it draws what the firmware said and stops short of the cell until the release event arrives. And when the machine cannot prove what physically happened, it locks and asks for a person rather than retrying.

The honest limits are equally clear, and they are limits of the hardware rather than of the approach. There is no emergency stop and no way to interrupt a motion. The steppers are open-loop and cannot detect a stall. The lens is tuned by eye and not calibrated. Placement supervision, the layer that would let the camera notice a block that never arrived, is fully designed and not built. Several measurements that would have made this report stronger were never taken, and they are marked as such and not filled in with plausible numbers.

6.2 Future Work

6.2.1 Safety, first

1. **A hardwired emergency stop.** A real path that removes hazardous motion energy independently of Python, USB, the event loop and firmware parsing: a latching mushroom button in series with a contactor on the 12 V motor rail, with a defined restart procedure. Every other item on this list is optional; this one is not.
2. **Make stop observable during motion.** The gantry's blocking loops would have to become a cooperatively serviced state machine, or poll a dedicated hardware stop input inside every pulse loop. A serial stop command alone is not enough, because the firmware does not read serial while it moves.
3. **A communications watchdog.** A lease from the Pi that the firmware requires to keep moving, so that a crashed master or a pulled cable brings the machine to a

defined safe state rather than leaving it to finish whatever it was told last. The requirement is that the safe state is genuinely safe, not merely a reset mid-carry.

4. **Electrical protection.** A fuse in the 12 V line sized to the supply, reverse-polarity protection at the input, and flyback protection on the servo rail.

6.2.2 Closing the loop on placement

The single most valuable feature that is designed and not built is **placement supervision**. The rig places a block and forgets it, so nothing in the system can notice when the board stops matching the plan. The detection half already exists and already labels every detection with an integer cell; three things are missing:

- **A server-side ledger** of what has been placed, which cell it went to and which level: the cumulative record that **last_result** and a selection cannot provide.
- **A rule for when it is safe to look.** The gantry occludes part of the surface, so a comparison made mid-cycle would report blocks missing that are simply behind the arm. The park state at the end of every build is the natural moment, and it is already a known, repeatable position.
- **A firmware verb that can retrieve a block from a cell.** **B** picks from the feeder and places; there is no pick-from-cell, so at present the machine can notice a wrong placement and can do nothing about it. A **P col row level** verb would make the supervision actionable rather than advisory.

6.2.3 Measurement and calibration

1. **A proper checkerboard or ChArUco lens calibration**, photographed at several positions and angles, replacing the by-eye tuning. This is what would turn the vision system from visually straight into measurement-grade, and it is a prerequisite for taking the camera's word about anything metric.
2. **Widen the saved calibration format** to carry a per-cell table instead of four envelope corners. That is the one change that removes the 0.27 cm mid-grid flattening error, and it touches every consumer of the format
3. **Fix the camera rather than the matrix.** The colour correction has reached its useful limit at 0.669 similarity; a lens with less veiling glare, or simply a longer lens further away, would do more than another iteration of the fit.

6.2.4 Protocol and software

1. **Controller-supplied command ids.** The Mega should echo an id the Pi chose rather than incrementing its own counter, and the Pi's waiter should reject a terminal acknowledgement that does not match. That single change removes the late-acknowledgement class of bug entirely and lets the prose fallback be deleted.

2. **A structured response for every command**, not only for **B**, **S**, **G**, **0** and **0+** are still completed by matching human prose and a quiet-line heuristic, which breaks the day someone rewords a message.
3. **One command arbiter for every public send**, so that a raw serial write cannot bypass the in-flight lock, plus an operating-system-level exclusive open so that a second process cannot silently take a board.
4. **Discard an overlong command line until the next terminator**, so that the tail of a too-long command can never become an executable command.

6.2.5 Capability

1. **Closed-loop or stall-detecting motion**. Encoders on X, Y and Z, or driver stall detection, would turn the one failure the machine cannot currently see into one it can. It would also allow the re-home before every move to be dropped, which is worth about a third of the cycle time.
2. **Painted blocks and a colour-sorting feeder**. The supply is currently bare wood in one uniform colour, and both the detector and the Studio already carry colour handling that nothing exercises: the Studio can assign a colour per block and preview which one must be staged next, while the feeder stages whatever is at the bottom of the hopper. Painting the supply and adding a sorting stage would turn colour into a real design dimension instead of unused capability.
3. **A faster serial link**. 9600 baud is the reason telemetry is limited to one line per phase. Raising it, in all three places that define it together, would allow throttled continuous position reporting and a twin that shows real motion and not phase transitions.
4. **A larger workspace, and a taller one**. The current envelope is a table-sized compromise. The frame stock and the CoreXY layout would both support a considerably larger machine, and the build ceiling of 25 cm is set by the Z travel rather than by anything fundamental.
5. **Plan projection onto the live camera**. Drawing the compiled program's remaining blocks onto the real camera image, in place, would make the design and the machine the same picture, and it is a small amount of work on top of a homography that already exists.
6. **Session evidence**. A timeline of every build with its timestamp, command, result, duration and a camera thumbnail; a time-lapse export made from a frame saved at every placement; and a run report exported as a document. All three are close to free, because the frames and the events already flow through the pipeline.

References

Sources are listed in the order they are first referred to in the report.

- [1] Greg06, "Automated Chessboard", Instructables, 1 March 2022.
<https://www.instructables.com/Automated-Chessboard/>. The CoreXY X/Y table and the limit-switch homing scheme this project's gantry is a modified version of.
- [2] G. Bradski, "The OpenCV Library", Dr. Dobb's Journal of Software Tools, 2000, and the OpenCV fisheye camera model documentation. The lens correction, the homography fit and the contour operations used by the block detector.
- [3] R. Hartley and A. Zisserman, "Multiple View Geometry in Computer Vision", 2nd ed., Cambridge University Press, 2004. The direct linear transform, its normalisation and the conditioning measure used as a calibration gate in Section 5.4.2.
- [4] A. B. Kahn, "Topological sorting of large networks", Communications of the ACM, 5(11), 1962. The ordering algorithm used by the Studio's compiler.
- [5] Toshiba, "TB6600 Stepper Motor Driver" application data, and Allegro MicroSystems, "A4988 DMOS Microstepping Driver with Translator and Overcurrent Protection". The gantry and feeder motor drivers.
- [6] Raspberry Pi Ltd., "Raspberry Pi 5 Product Brief" and "Picamera2 Library Manual". The master controller and the only supported camera interface on the Pi 5's CSI bus.

Appendices

Appendix A: Bill of Materials

Table 21: Compute and control.

Item	Qty	Note
Raspberry Pi 5, 8 GB RAM	1	Master controller
Arduino MEGA 2560	1	Gantry controller
Arduino Uno	1	Feeder controller
Raspberry Pi camera module, OV5647, 160 deg fisheye, + ribbon	1	DORHEA module

Table 22: Motion, actuators and sensors.

Item	Qty	Note
NEMA17 stepper motor	4	2 CoreXY, 1 Z, 1 feeder belt
TB6600 stepper driver	3	The three gantry axes
A4988 stepper driver	1	Feeder belt
28BYJ-48 stepper motor, 5 V	1	Claw rotation
ULN2003 driver board	1	For the 28BYJ-48
Hobby servo	3	Gripper, container gate, aligner
Micro limit switch	4	X home, Y home, Z bottom, Z top
HC-SR04 ultrasonic sensor	2	Container exit, pickup stage

Table 23: Mechanical.

Item	Qty	Note
Linear rail + linear motion bearing	1	1.5 m x 15 mm; carries the Z axis
Aluminium profile	3.6 m	Cut from 6 m stock: 4 legs at 15 cm, 3 members along X at 30 cm, 2 along Y at 60 cm, 2 for the feeder at 30 cm, 1 for the Z column at ~30 cm
GT2 timing pulley	~9	Drive and idler pulleys, CoreXY and Z
GT2 timing belt	5 m	Plus 1 m of additional belt
Belt sheet	1	The conveyor surface
Castor wheel	8	X/Y guidance, on 3D-printed carriages
3D-printed parts	many	Carriages, the claw, the conveyor parts, mounts
Wooden blocks, 2.2 x 6.0 x 1.5 cm	30	Bare wood, one uniform colour

Table 24: Electrical and power.

Item	Qty	Note
Switched-mode power supply, 12 V / 15 A	1	Motor rail
LM2596 adjustable buck converter, 4-40 V in, 1.25-37 V out, 3 A, with display	1	Set to 5 V
Raspberry Pi 5 official USB-C power supply	1	The Pi only
Resistors and capacitors, assorted	1 lot	A4988 current reference and local decoupling
Connectors, cropped M/M jumper leads, heat-shrink	1 lot	All signal wiring

Appendix B: Pin Mappings

Both tables are taken directly from the firmware sources and not from any documentation of them.

Table 25: Arduino MEGA 2560 gantry controller: pin map.

Pin	Direction	Connected to	Function
2	output	TB6600 #1 DIR	Motor 1 direction (CoreXY)
3	output	TB6600 #1 STEP	Motor 1 step pulse
4	output	TB6600 #1 EN	Motor 1 enable, active LOW
6	output	Gripper servo signal	Claw jaws: OPEN 0 deg, CLOSE 52 deg
8	output	TB6600 #2 DIR	Motor 2 direction (CoreXY)
9	output	TB6600 #2 STEP	Motor 2 step pulse
10	output	TB6600 #2 EN	Motor 2 enable, active LOW
11	output	TB6600 #3 DIR	Z direction
12	output	TB6600 #3 STEP	Z step pulse (no enable line fitted; the driver is permanently enabled)
28	input, pull-up	Z bottom limit switch	Z home / GROUND reference, NC
29	input, pull-up	Z top limit switch	Z far-end stop, NC
30	input, pull-up	X limit switch	X home / zero, NC
31	input, pull-up	Y limit switch	Y home / zero, NC
36	output	ULN2003 IN2 (green)	Claw rotation stepper
37	output	ULN2003 IN4 (red)	Claw rotation stepper
38	output	ULN2003 IN1 (black)	Claw rotation stepper
39	output	ULN2003 IN3 (blue)	Claw rotation stepper
USB	serial	Raspberry Pi	9600 8N1

Table 26: Arduino Uno feeder controller: pin map.

Pin	Direction	Connected to	Function
2	output	A4988 DIR	Belt direction
3	output	A4988 STEP	Belt step pulse (ENABLE is tied to ground)
4	output	Exit HC-SR04 TRIG	Container-exit sensor trigger
5	input	Exit HC-SR04 ECHO	Container-exit sensor echo
6	output	Alignment servo signal	Rest 90 deg, nudge 120 deg
8	output	Stage HC-SR04 TRIG	Pickup-point sensor trigger
9	input	Stage HC-SR04 ECHO	Pickup-point sensor echo
12	output	Container servo signal	Closed 20 deg, stage 1 at 90 deg, open 160 deg
USB	serial	Raspberry Pi	9600 8N1, protocol 2

Appendix C: Serial Protocols

C.1 Gantry (MEGA) acknowledgement lines

```
@<seq> <KIND> [reason text | key=value ...]

@0 BOOT fw=build_test_v1
@0 READY grid=6x5 mode=vertical
@n RECV cmd=B col=3 row=5 level=0
@n STEP step=8 total=14 phase=move_to_target action=move \
      text=Move_XY_to_the_target_cell status=begin
@n STEP step=11 total=14 phase=release action=release \
      text=Open_the_claw_and_release status=done
@n ERR expected: B <col> <row> <level>
@n SAFE cell out of range
@n HELD Z never reached the ground switch
@n OK col=3 row=5 level=0
```

OK, **ERR**, **SAFE** and **HELD** are terminal; **BOOT**, **READY**, **RECV** and **STEP** are not. Sequence 0 means nobody asked. Every ack literal is stored in flash rather than SRAM. A **BUSY** kind is reserved and understood by the Pi's parser but is not emitted, because the firmware runs one command at a time and never sees a second.

C.2 A complete build transaction

```
@12 RECV cmd=B col=3 row=5 level=0
@12 STEP step=1 total=14 phase=raise_clear action=move status=begin ms=2570
@12 STEP step=2 total=14 phase=home_feeder action=move status=begin
@12 STEP step=3 total=14 phase=neutralise_claw action=rotate status=begin
@12 STEP step=4 total=14 phase=open_claw action=release status=begin
@12 STEP step=5 total=14 phase=lower_to_ground action=move status=begin ms=2570
@12 STEP step=6 total=14 phase=grip action=grip status=begin
@12 STEP step=7 total=14 phase=lift_block action=move status=begin ms=2570
@12 STEP step=8 total=14 phase=move_to_target action=move status=begin
@12 STEP step=9 total=14 phase=rotate_to_grid action=rotate status=begin
@12 STEP step=10 total=14 phase=lower_to_level action=move status=begin ms=2386
@12 STEP step=11 total=14 phase=release action=release status=begin
@12 STEP step=11 total=14 phase=release action=release status=done
@12 STEP step=12 total=14 phase=park_clear action=park status=begin ms=2570
@12 STEP step=13 total=14 phase=park_home action=park status=begin
@12 STEP step=14 total=14 phase=park_rotation action=park status=begin
@12 OK col=3 row=5 level=0
```

The **text=** field is omitted above for width; on the wire every **STEP** line also carries the underscored human label.

C.3 Feeder (Uno) protocol 2

```
@0 READY firmware=belt_v1 protocol=2 board=uno

> FEED 42
@42 RECV cmd=FEED
@42 SENSOR sensor=stage distance_cm=32.6 detected=0
@42 ACK cmd=FEED accepted=1
@42 STATE state=closing
@42 EVENT phase=container_closing
@42 STATE state=opening_stage_1
@42 STATE state=opening_stage_2
@42 STATE state=waiting_for_exit
@42 STATE state=moving_to_stage
@42 SENSOR sensor=exit distance_cm=7.4 detected=1
@42 EVENT phase=exit_detected_container_closed_belt_running distance_cm=7.4
@42 STATE state=aligning
@42 SENSOR sensor=stage distance_cm=8.2 detected=1
@42 STATE state=verifying_stage
@42 STATE state=block_ready
@42 EVENT phase=block_ready distance_cm=8.1
@42 OK state=block_ready result=staged
```

Only the final **OK** is permission to pick the block up. **ACK**, **STATE**, **SENSOR** and **EVENT** are progress telemetry. The four terminal failures are **stage_occupied**, **exit_timeout**, **stage_timeout** and **cancelled**.

Appendix D: Firmware Constants

Table 27: The gantry firmware's physical constants. These live only in the sketch and are deliberately not copied into the Pi's configuration, because nothing can push them over serial.

Constant	Value	Meaning
SOFT_LIMIT_X_TRAVEL	4,550 steps	X travel cap
SOFT_LIMIT_Y_TRAVEL	7,600 steps	Y travel cap

Constant	Value	Meaning
X_TRAVEL_CM	22.8 cm	Measured holder displacement over that cap
Y_TRAVEL_CM	38.0 cm	The same, for Y
Z_TRAVEL_STEPS	1,350 steps	Switch to switch; calibration only
Z_TRAVEL_CM	26.5 cm	Switch to switch, tape-measured
BLOCK_HEIGHT_CM	1.5 cm	One stack level
MAX_BUILD_HEIGHT_CM	25.0 cm	Build ceiling; highest level 16
Z_MARGIN_FIXED_CM	+0.10 cm	Added once at any level ≥ 1
Z_MARGIN_PER_LEVEL_CM	0.00 cm	Cumulative per-level trim
STEP_DELAY / STEP_DELAY_Z	575 us / 950 us	Step half-periods
DIR_SETTLE_MS	5 ms	After a direction change
LIMIT_CONFIRM_US	200 us	Switch confirmation time
SERVO_OPEN_ANGLE / SERVO_CLOSE_ANGLE	0 deg / 52 deg	Claw jaws
SERVO_SETTLE_MS	600 ms	After every commanded servo move in a build
AUX_STEPPER_STEPS_PER_REV	2,048	28BYJ-48 output revolution
AUX_STEPPER_SPEED_RPM	10	Claw rotation speed
BUILD_PHASE_PAUSE_MS	250 ms	Settle between build phases
BUILD_STEP_COUNT	14	Phases per build
SKEW_Y_PER_COL_CM	0.1 cm	X-rail skew compensation, build motion only
TOOL_OFFSET_CW_*	(+0.9, -0.3) cm	The pickup-rotate swing
EN_ACTIVE_LEVEL	LOW	TB6600 enable polarity in this wiring

Appendix E: Grid Geometry

The lattice, in one line, for both modes:

```

pitch      = block + gap
centre(i) = trim + error_offset + shift + i * pitch

holder target = desired block centre - tool offset(rotation)

target_cm   = level * (BLOCK_HEIGHT_CM + Z_MARGIN_PER_LEVEL_CM) + Z_MARGIN_FIXED_CM
target_steps = round(target_cm * zStepsPerCm()) + Z_MARGIN_FIXED_STEPS

yNudge_cm = SKEW_Y_PER_COL_CM * col      (build motion only)

```

Table 28: Complete grid geometry, both modes, at the shipped calibration.

Mode	Axis	Block (cm)	Gap (cm)	Pitch (cm)	Trim (cm)	Cells	Footprint (cm)	Centres (cm)	Block edges (cm)
vertical	X	2.2	1.6	3.8	0.0	7	25.00	0.00 - 22.80	-1.10 - 23.90
vertical	Y	6.0	1.6	7.6	0.0	6	44.00	0.00 - 38.00	-3.00 - 41.00
horizontal	X	6.0	1.6	7.6	+1.9	3	21.20	1.90 - 17.10	-1.10 - 20.10

Mode	Axis	Block (cm)	Gap (cm)	Pitch (cm)	Trim (cm)	Cells	Footprint (cm)	Centres (cm)	Block edges (cm)
horizontal	Y	2.2	1.6	3.8	+1.9	10	36.40	1.90 - 36.10	0.80 - 37.20

Cell counts are counts; the firmware's **S** command and its internal tables speak in highest indices, which is one less. Cell [0,0] is the feeder in both modes and is never built on, so the buildable counts are 41 and 29. Edge overhang budgets are 1.1 / 3.0 cm for vertical and 3.0 / 1.1 cm for horizontal, and they are what the geometry check measures the block edges against; they move nothing.

Appendix F: Experimental Data

The build-timing data analysed in Chapter 5 is summarized in Table 18 (per-phase statistics), while the five-level stacking run is reported in Table 19. The raw sources are the two append-only log files the service writes on every run:

logs/build.log: One clearly separated section per build: the request, the job handoff, the board's **RECV**, every firmware phase with the ETA the firmware predicted beside the time the phase actually took, and the settled result with the total elapsed. Every timestamp in a section is relative to that build's start, so the section reads as a stopwatch.

logs/serial.log: Every line to and from either Arduino, each stamped with the wall clock and the gap since the previous serial line, tagged **[MEGA/GANTRY]** or **[UNO/FEEDER]**. A stall on the cable or a slow phase shows up directly as a large delta in the second column.

One complete build section, verbatim from **logs/build.log**:

```
BUILD 2026-09-03 13:53:14 B 3 2 4
selection=(3, 2) level=4 mode=vertical
+0.00s request accepted by /api/build
+0.00s build job thread started
+0.14s board RECV seq=5
+0.41s phase 1/14 begin raise_clear [move] firmware-ETA 0.19s
      phase 1 raise_clear took 0.43s (firmware ETA 0.19s, +0.24s)
+0.84s phase 2/14 begin home_feeder [move]
      phase 2 home_feeder took 0.58s
+1.41s phase 3/14 begin neutralise_claw [rotate]
      phase 3 neutralise_claw took 0.41s
+1.82s phase 4/14 begin open_claw [release]
      phase 4 open_claw took 0.93s
+2.75s phase 5/14 begin lower_to_ground [move] firmware-ETA 2.39s
      phase 5 lower_to_ground took 2.91s (firmware ETA 2.39s, +0.53s)
+5.66s phase 6/14 begin grip [grip]
      phase 6 grip took 0.95s
+6.62s phase 7/14 begin lift_block [move] firmware-ETA 2.57s
      phase 7 lift_block took 2.89s (firmware ETA 2.57s, +0.32s)
+9.51s phase 8/14 begin move_to_target [move]
      phase 8 move_to_target took 7.12s
+16.63s phase 9/14 begin rotate_to_grid [rotate]
      phase 9 rotate_to_grid took 0.39s
+17.02s phase 10/14 begin lower_to_level [move] firmware-ETA 1.80s
      phase 10 lower_to_level took 2.26s (firmware ETA 1.80s, +0.45s)
+19.28s phase 11/14 begin release [release]
      phase 11 release took 0.71s
+19.99s phase 11/14 release confirmed
+20.12s phase 12/14 begin park_clear [park] firmware-ETA 1.99s
      phase 12 park_clear took 2.25s (firmware ETA 1.99s, +0.26s)
+22.37s phase 13/14 begin park_home [park]
      phase 13 park_home took 7.04s
+29.41s phase 14/14 begin park_rotation [park]
      phase 14 park_rotation took 0.43s
+29.84s RESULT PLACED
+29.84s build finished, total 29.84s
```