

Assignment 4: Two Stops, One Brain — Teacher Guide

At a glance

Learning objective	THE keystone lesson: run a multi-step plan inside a program that re-runs every second. Students build a two-stop route using a step variable that survives from second to second, plus <code>if ... then ...</code> blocks that act on the current step and advance it on arrival: the state-machine pattern, in blocks.
Time estimate	60–90 minutes
Project setup	New (or continued) project on the AsteroidBee 2026 game, Graphical Editor
Prerequisite assignment	Assignments 1–3
Blocks introduced	Scalar declare block type: ▼ name: <code>__</code> initial value: <code>__</code> (Variables palette, init page only, type int); the variable get block (it shows just the name, e.g. <code>step</code>) and the variable set block (<code><name> = [value]</code>) for a single variable. Everything else carries over from Assignment 3.

The one idea this assignment teaches

Assignment 2 ended with a hard truth: stacking two `setPos __, __, __` blocks does **not** make two stops. Both blocks run within the same second, the second one overwrites the first, and the satellite flies only to the last target. The main page is not a story that plays out over time; it is a checklist the satellite re-reads every second. So how does any program ever do step one, then step two?

The answer is **memory plus questions**. A variable declared on the init page keeps its value from one second to the next: it is the program's notebook. If that variable (call it `step`) says `0`, the program does the stop-A blocks; if it says `1`, the stop-B blocks. And the moment the satellite actually arrives at a stop, an `if ... then ...` block changes the variable, so the **next** second, a different branch of the page wins. The plan unfolds over time even though the page itself never changes.

🕒 **Every second:** the main page runs top to bottom. Read my position; ask "is `step 0`?" (if yes: fly toward A, and check whether I've arrived); ask "is `step 1`?" (if yes: fly toward B, and check arrival there). Only one of those questions is true on any given second, and the `step` variable is what carries the answer from this second into the next one.

Engineers call this a state machine, but the class only needs three words: **remember the step, act on the step, advance the step when its goal is met**. Every later assignment (spy photos, energy runs, the full match) is this same pattern with more steps.

⚠ **Engine note — "parking" still means commanding.** In the ZR engine a `setPos` command lasts exactly one second; the satellite holds a spot only on the seconds the program actually runs a `setPos` block (re-issuing the same destination every second is how it stays put — that is normal, not wasteful). So a step with **no** branch does not "park": once nothing runs `setPos`, the satellite drifts off on its residual motion, because microgravity has no friction. That is why the route's final step is a real `step == 2` branch with a `setPos __, __, __` to B inside it, **not** an empty step. Symptom to recognize: the satellite reaches B, pauses, then slowly wanders away — often first noticed around the $t = 60$ s light switch, even though the light switch applies no force and is unrelated.

Before class

- [] Re-read [how-blocks-run.md](#): this assignment is that document, made physical.
- [] Build the reference solution yourself once (15–20 minutes); simulate it so you know what a correct run looks and sounds like.
- [] Decide: will students **reuse their Assignment 3 project** (recommended, since the `myState` array declare and the arrival check are already built) or start a fresh Graphical Editor project on AsteroidBee 2026 and rebuild the array declare?
- [] Have the arena diagram ready to project or print (`../images/arena-diagram.svg`) so you can mark Stop A and Stop B on it.
- [] Find the **Variables** palette category and the **init** page tab in the editor yourself, so you can point at them without hunting.
- [] Print [printable.md](#) for each student if you want a paper packet.

Walkthrough: what to say and do

1. **Open with the failed idea.** Say: "In Assignment 2 you proved that two stacked `setPos` `_`, `_`, `_` blocks do NOT make two stops. The satellite only obeyed the last one. Today we fix that. Not with more movement blocks, but with memory."
2. **The plan on paper.** Draw the arena (or project `../images/arena-diagram.svg`). Mark **Stop A = (0, -0.4)** and **Stop B = (0.4, -0.4)**. Ask the class: "The page re-runs every second. What does the satellite need to KNOW each second to run this route?" Steer them to: it needs to know **which stop it is working on right now**.
3. **Give it memory: the init page.** Switch to the **init** page. From the Variables palette, drag a declare block, type: `▼` name: `_` initial value: `_`, into the **"global variables"** slot, underneath the `myState` array declare from Assignment 3 (or rebuild that array declare first: type `float`, name `myState`, length 12). Set the new one to type **int**, name **step**, initial value **0**. Say: "This page runs once, at the start of the match. A variable declared here survives from second to second. This is the program's memory, its notebook."
4. **The shape first, in words.** Before dragging anything on the main page, write this on the board: Every second: read my position. IF I am on step 0: fly to A, and if I have ARRIVED at A, set step to 1. IF I am on step 1: fly to B, and if I have arrived at B, set step to 2 and celebrate. Point out there are **two big if s**, and each one **owns** its stop, including that stop's arrival test.
5. **Build it.** On the main page, in the "loop" slot, top to bottom:
 6. `get My ZRState` with the `myState` array attached (same as Assignment 3).
 7. An `if ... then ...` block whose question is the compare block `[a] == ▼ [b]` holding `get step` and the number `0`. Inside it: `setPos` `_`, `_`, `_` with **0, -0.4, 0**, and then the nested arrival-`if` from Assignment 3, built from `[a]` and `▼ [b]` joining two compares: `absolute value of myState [ 0 ]` is less than `0.05`, and `absolute value of (myState [ 1 ] + 0.4)` is less than `0.05`. Inside that arrival-if: `set step = [value]` with **1**, and `DEBUG [text]` with the text block `""` reading **"Checkpoint A!"**.
 8. A second `if ... then ...` below the first, asking `step == 1`. Inside: `setPos` `_`, `_`, `_` with **0.4, -0.4, 0**, and a nested arrival-if for B checking `absolute value of (myState [ 0 ] - 0.4)` less than `0.05`, **and** `absolute value of (myState [ 1 ] + 0.4)` less than `0.05`. Inside it: `set step = [value]` with **2**, and `DEBUG` with **"Route complete!"**.
 9. A **third** `if ... then ...` below those two, asking `step == 2`. Inside it, just `setPos` `_`, `_`, `_` with **0.4, -0.4, 0** again — no arrival-if, no `DEBUG`. This is the "park" branch (see the engine note above): it keeps re-issuing B's destination every second so the satellite actually holds there instead of drifting off.
10. **Trace one second aloud, three moments.** Walk the page with your finger:

11. Second 5: "step is 0. First `if`: true. Destination A is set, but I'm not within 5 cm yet, so the arrival-if does nothing. Second `if`: `step = 1`? No. Done. Next second, same again."
12. The second of arrival at A: "First `if`: true. Arrival-if: TRUE. It fires **once**: step becomes 1, 'Checkpoint A!' prints. Note: second `if` asks `step = 1` and that's now true, so stop B's `setPos` takes over this same second."
13. The second after: "First `if`: `step = 0`? **No**. That whole branch is now skipped, forever. Second `if`: yes, flying to B."
14. After arriving at B: "Arrival-if fires once: step becomes 2, 'Route complete!' prints. Now the second `if` (`step = 1`?) is false too, and the **third** `if` (`step = 2`?) is true, so `setPos` to B runs every second from here on — that is what keeps the satellite parked at B instead of drifting off."
15. **Simulate**. Run it (as Blue). Watch the viewer: flight to A, a brief stop, then on to B and park. Let the class narrate which `if` is "winning" as it flies.
16. **The contrast with Assignment 3**. Point at the log: "In Assignment 3 your arrival message printed every single second once you arrived, because the page kept re-running and the arrival test kept being true. Today 'Checkpoint A!' printed exactly ONCE. Why?" Answer to draw out: the same second that prints it also changes `step`, so next second, the question `step = 0` is false and the whole branch is skipped.
17. **Show the C**. Flip the **C Code** toggle. Students should recognize the shape: the two declares above, `step = 0;` inside `init()`, and **two if blocks inside** `loop()`, the function the satellite runs once per second.

The mistakes you will see

Mistake 1: No step variable at all. (The "blocks run in sequence over time" confusion, back again, now at the plan level.)

- **What it looks like:** Two `setPos` `_`, `_`, `_` blocks, each wrapped in its own arrival-if, but nothing anywhere remembering progress. The satellite flies only to the last target: Assignment 2's lesson resurfacing.
- **Why it happens:** Students believe wrapping each stop in an `if` is enough to make them take turns. But both ifs are asked fresh every second, and nothing records that stop A was ever finished.
- **The question to ask:** "When the page re-runs next second, how does the satellite know it already finished stop A?"

Mistake 2: Resetting the memory every second.

- **What it looks like:** A `set step = [value]` block with 0 dragged to the TOP of the main page, "to make sure it starts at 0." The satellite never leaves stop A.
- **Why it happens:** The student is thinking "initialize first, then run" as a one-time sequence. But the main page re-runs every second, so the reset runs every second too, wiping the notebook every second, an instant after the arrival-if wrote in it.
- **The question to ask:** "Which page should something-that-happens-once live on?" (The init page: that is exactly what the declare block's initial value is for.)

Mistake 3: Advancing without an arrival check, or nesting the wrong arrival test inside the wrong step.

- **What it looks like:** Either the satellite "gives up" on A halfway (the `set step = [value]` sits outside the arrival-if, so it runs in the very first second), or "Route complete!" fires at stop A (B's arrival test got nested inside step 0's `if`, where A's coordinates make it true, or vice versa).
- **Why it happens:** With two levels of nested ifs, blocks get dropped one slot off from where the student intended, and the page still compiles fine.
- **The question to ask:** "Which step OWNS this arrival test? Is it inside that step's `if`?"

Reference solution at a glance

- **init page:** two declare blocks in the "global variables" slot, the `myState` float array (length 12) from Assignment 3, and below it an **int** named `step` with initial value **0**.
- **main page:** in the "loop" slot, top to bottom: `get My ZRState into myState`; an `if ... then ... for step = 0` containing `setPos __, __, __ (0, -0.4, 0)` and a nested arrival-if that does `set step = [value] (1)` and `DEBUG "Checkpoint A!"`; a second `if ... then ... for step = 1` containing `setPos __, __, __ (0.4, -0.4, 0)` and a nested arrival-if that does `set step = [value] (2)` and `DEBUG "Route complete!"`; and a third `if ... then ... for step = 2` containing just `setPos __, __, __ (0.4, -0.4, 0)` to hold at B (no arrival-if, no `DEBUG`).

The full layout and the exact generated C are in [reference-solution.md](#).

Wrap-up questions

1. In Assignment 3 the arrival message printed every second forever. Today "Checkpoint A!" printed exactly once, yet the page that prints it still runs every second. What changed?
2. Suppose we want a third stop, C = (0.4, 0). Exactly which blocks would you add, and would any existing block need to change? (Answer: insert an `if step = 2` branch that flies to C and, on arrival, sets `step = 3`; then change the final hold branch's question from `step == 2` to `step == 3`. The two earlier travel branches are unchanged.)
3. What would happen if you deleted just the `set step = [value]` block inside stop A's arrival-if? Predict the viewer **and** the log before you try it.