# SPIMbot documentation

To make some of our labs a little more interesting, we’ll write programs for NASA’s next-generation Mars rover called SPIMbot. Due to budget cuts, SPIMbot is not nearly as sophisticated as previous Mars rovers.

This year, SPIMbot must move near a star, mining asteroids and dropping them off at its space station (only in part2) to score points.

SPIMbot must deal with several limitations, including a cargo limit and gravity, while scoring points.

## Error rendering macro ‘toc’


## SPIMbot Input/Output

SPIMbot's sensors and controls are manipulated via memory-mapped I/O; that is, the I/O devices are queried and controlled by reading and writing particular memory locations. All of SPIMbot's I/O devices are mapped in the memory range 0xffff0000 - 0xffffffffff. Below we describe SPIMbot's I/O devices and their associated control and data registers. Crossed out I/Os are disabled for now and will be enabled for the competition later.

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Acceptable Values/Range</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>VELOCITY</td>
<td>0xffff0010</td>
<td>5 to -5</td>
<td>Gives your SPIMbot's current velocity.</td>
<td>Immediately updates your SPIMbot's velocity.</td>
</tr>
<tr>
<td>ANGLE</td>
<td>0xffff0014</td>
<td>-360 to 360</td>
<td>Gives your SPIMbot's current orientation.</td>
<td>No immediate effect; the value written is used when A NGLE_CONTROL is written.</td>
</tr>
<tr>
<td>ANGLE_CONTROL</td>
<td>0xffff0018</td>
<td>0 (relative), 1 (absolute)</td>
<td>N/A</td>
<td>Updates your SPIMbot's orientation, using the last value written to ANGLE as a relative or absolute angle (depending on the value written to this address).</td>
</tr>
<tr>
<td>TIMER</td>
<td>0xffff001c</td>
<td>0 to 0xffffffff</td>
<td>Gives the number of elapsed cycles.</td>
<td>Requests a timer interrupt at the cycle written.</td>
</tr>
<tr>
<td>BOT_X</td>
<td>0xffff0020</td>
<td>0 to 300</td>
<td>Gives your SPIMbot's current X location.</td>
<td>N/A</td>
</tr>
<tr>
<td>BOT_Y</td>
<td>0xffff0024</td>
<td>0 to 300</td>
<td>Gives your SPIMbot's current Y location.</td>
<td>N/A</td>
</tr>
<tr>
<td>OTHER_BOT_X</td>
<td>0xffff00a0</td>
<td>N/A</td>
<td>Gives your opponent SPIMbot's X location.</td>
<td>N/A (NOTE: THIS IS NOT USED IN LAB 10)</td>
</tr>
<tr>
<td>OTHER_BOT_Y</td>
<td>0xffff00a4</td>
<td>N/A</td>
<td>Gives your opponent SPIMbot's Y location.</td>
<td>N/A (NOTE: THIS IS NOT USED IN LAB 10)</td>
</tr>
<tr>
<td>SCORES_REQUEST</td>
<td>0xffff1018</td>
<td>any valid data address</td>
<td>N/A</td>
<td>Writes the score information to the provided address. The score information will be an array of 2 integers where your score is the first element.</td>
</tr>
<tr>
<td>REQUEST PUZZLE</td>
<td>0xffff00d0</td>
<td>any valid data address</td>
<td>N/A</td>
<td>Queues a request for a new puzzle and will eventually store the puzzle at the given data address. Only after a delay will the puzzle be written to the address. At the end of this delay, a REQUEST PUZZLE interrupt will be delivered. See below for more details (NOTE: THIS IS NOT USED IN LAB 10).</td>
</tr>
<tr>
<td>SUBMIT SOLUTION</td>
<td>0xffff00d4</td>
<td>any valid data address</td>
<td>N/A</td>
<td>Submit your puzzle solution. See below for more details.</td>
</tr>
</tbody>
</table>
We describe these in more detail below.

**Orientation Control**

SPIMBot's orientation can be controlled in two ways: 1) by specifying an adjustment relative to the current orientation, and 2) by specifying an absolute orientation.

In both cases, an integer value (between -360 and 360) is written to `ANGLE` (0xffff0014) and then a command value is written to the `ANGLE_CONTROL` (0xffff0018). If the command value is 0, the orientation value is interpreted as a relative angle (i.e., the current orientation is adjusted by that amount). If the command value is 1, the orientation value is interpreted as an absolute angle (i.e., the current orientation is set to that value).

Angles are measured in degrees, with 0 defined as facing right. Positive angles turn SPIMBot clockwise. While it may not sound intuitive, this matches the normal Cartesian coordinates (in that the +x direction is angle 0, +y=90, -x=180, and -y=270), since we consider the top-left corner to be (0,0) with +x and +y being right and **down**, respectively.

**Bonk (Wall Collision) Sensor**

The bonk sensor signals an interrupt whenever SPIMBot runs into a wall. Note that SPIMBot's velocity is set to zero when it hits a wall. (See below how to register for and acknowledge interrupts.)
Also note that SPIMbot can never hit the left wall.

**Timer**

The timer does two things: 1) the number of cycles elapsed can be read from the `TIMER` memory-mapped I/O address, and 2) a timer interrupt can be requested by writing the cycle number at which the interrupt is desired.

**Asteroid Map**

The asteroid map will give you the location and points for each asteroid on the map. You should reserve space in your `.data` section to hold the contents of the map, and then give the address of the reserved space to `ASTEROID_MAP` to cause the map to be written to your data segment. The map is a combination of two structs:

```c
struct Asteroid {
    short y;
    short x;
    int points;
};

struct AsteroidMap {
    unsigned length; // store the number of asteroids
    struct Asteroid asteroids[50]; // hold data for each asteroid
};
```

The asteroid map will be written to the address written to the IO. Note that this map might not always have 50 asteroids; the length variable holds the number of asteroids actually represented in the map. Despite this, these data structures will always take up a constant amount of memory.

If you need a hint on how to deal with the shorts in asteroid, see the next section.

**Station Location**

To get the station location, you must read from the `STATION_LOC` IO address. This will return the station position in the following form:

```
31 16 15 0
0101110000111011 1101000100110100
```

You must extract each component individually using shifting and masking expressions.

**Puzzle**

The puzzle is the same puzzle as from Lab8 so feel free to go back to that lab's handout for more details. As a summary, your bot will be given a Puzzle struct. The Puzzle struct is made up of a Canvas struct, a Line struct, and a data array. Your goal is to figure out the number of disjoint regions after each line is drawn onto the canvas. This puzzle is therefore just a re-skin of lab 8.
Note that if you are making your own puzzle solver, the -debug will printout helpful info about why your solution was incorrect.

**Requesting a Puzzle**

The first step to getting a puzzle is to make a request for one using the REQUEST_PUZZLE memory I/O. However, it takes some time to generate a puzzle, so the puzzle will not be generated immediately. Instead, you will have to wait for a REQUEST_PUZZLE interrupt. Note that you must enable the REQUEST_PUZZLE interrupt or else you will never receive a puzzle. Right before the interrupt is delivered, the puzzle will be written to the address given to the REQUEST_PUZZLE memory I/O, and afterwards the interrupt will be delivered.

As a side note, you can request another puzzle before you've solved the current one. When you submit the solution, it will be compared to the oldest puzzle that hasn't had a solution submitted for yet. You can have as many outstanding puzzles as you want.

**Puzzle Struct**

The format the puzzles come in is via a struct which contains the information of Canvas and Lines. The maximum canvas size is 12×12. The maximum number of lines is 12. The struct written to the bot memory is defined as follows:

```c
struct Puzzle {
    Canvas canvas;
    Lines lines;
    char data[300];
};
```

**Submitting Your Solution**

In order to solve the puzzle using existing lab 8 solutions, it is necessary to create your own Solution struct in MIPS and then pass that to the count_disjoint_regions function call after setting up the struct correctly.

To submit your solution, you will simply write the address of the array in the solution struct from count_disjoint_regions to the SUBMIT_SOLUTION mmio address. If your solution is correct, you will be rewarded with 200 energy.

**Throwing a Puzzle**

If you write to the THROW_PUZZLE address before you write to SUBMIT_SOLUTION, your puzzle will be "thrown" to the other bot, freezing it until it can solve it.

However, you should use these throws carefully! You only get 10 throws - if you try to throw after that, you will get the energy reward instead.

**Interrupts**

The MIPS interrupt controller resides as part of co-processor 0. The following co-processor 0 registers (which are described in detail in section A. 7 of your book) are of potential interest:

<table>
<thead>
<tr>
<th>Name</th>
<th>Register</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Register</td>
<td>$12</td>
<td>This register contains the interrupt mask and interrupt enable bits.</td>
</tr>
<tr>
<td>Cause Register</td>
<td>$13</td>
<td>This register contains the exception code field and pending interrupt bits.</td>
</tr>
<tr>
<td>Exception Program Counter (EPC)</td>
<td>$14</td>
<td>This register holds the PC of the executing instruction when the exception/interrupt occurred.</td>
</tr>
</tbody>
</table>
Interrupt Acknowledgment

At the end of handling an interrupt, it is important to notify the device that its interrupt has been handled, so that it can stop requesting the interrupt. This process is called "acknowledging" the interrupt. As is usually the case, interrupt acknowledgment in SPIMbot is done by writing to a memory-mapped I/O location.

In all cases, writing the acknowledgment addresses with any value will clearing the relevant interrupt bit in the Cause register, which enables future interrupts to be detected.

<table>
<thead>
<tr>
<th>Name</th>
<th>Interrupt Mask</th>
<th>Acknowledge Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>0x8000</td>
<td>0xffff006c</td>
</tr>
<tr>
<td>Bonk (wall collision)</td>
<td>0x1000</td>
<td>0xffff0060</td>
</tr>
<tr>
<td>Request Puzzle</td>
<td>0x800</td>
<td>0xffff00d8</td>
</tr>
<tr>
<td>Station Enter Interrupt</td>
<td>0x400</td>
<td>0xffff0058</td>
</tr>
<tr>
<td>Station Exit Interrupt</td>
<td>0x2000</td>
<td>0xffff0064</td>
</tr>
<tr>
<td>Bot Freeze Interrupt</td>
<td>0x4000</td>
<td>0xffff00e4</td>
</tr>
</tbody>
</table>

Request Puzzle

You will receive the REQUEST_PUZZLE interrupt once the requested puzzle is written into the provided memory address. The puzzle is the same puzzle as from Lab 8 so feel free to go back to that lab's handout for more details.

Station Enter Interrupt

This interrupt will happen immediately after your station enters the accessible area of the map.

Station Exit Interrupt

This interrupt will happen immediately after your station exits the accessible area of the map.

Bot Freeze Interrupt

This interrupt will happen immediately after your opponent throws a puzzle at you. Once the puzzle is thrown, the engine velocity of your bot will be set to 0, so the bot is frozen. To unfreeze the bot, you first need to acknowledge the interrupt with a valid address to hold the puzzle thrown by the other bot. Unlike REQUEST_PUZZLE, the provided address will be populated with a puzzle with no delay. You can then solve the puzzle and submit the solution to UNFREEZE_BOT. The velocity of your bot will be restored after submitting the solution.

Running and Testing Your Code

QtSpimbot's interface is much like that of QtSpim (upon which it is based). You are free to load your programs as you did in QtSpim using the buttons. Both QtSpim and QtSpimbot allow your programs to be specified on the command line using the -file argument. Be sure to put other flags before the -file flag.

The -debug flag can be very useful and will tell QtSpimbot to print out extra information about what is happening in the simulation. You can also use the -drawcycles flag to slow down the action and get a better look at what is going on.

In addition, QtSpimbot includes two arguments (-maponly and -run) to facilitate rapidly evaluating whether your program is robust under a variety of initial conditions (these options are most useful once your program is debugged).

During the tournament, we'll run with the following parameters:

- -maponly
- -run
- -tournament
- -randommap
- -largemap
- -exit_when_done

Is your QtSpimbot instance running slowly? Try selecting the "Data" tab instead of the "Text" one.

Are you on Linux and having theming issues? Try adding -style d to your command line arguments.
# Useful Command Line Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-file &lt;file1.s&gt; &lt;file2.s&gt; ...</td>
<td>Specifies the assembly file(s) to use</td>
</tr>
<tr>
<td>-file2 &lt;file1.s&gt; &lt;file2.s&gt; ...</td>
<td>Specifies the assembly file(s) to use for a second SPIMbot</td>
</tr>
<tr>
<td>-part1</td>
<td>Run SPIMbot under Lab 10 part 1 conditions</td>
</tr>
<tr>
<td>-part2</td>
<td>Run SPIMbot under Lab 10 part 2 conditions</td>
</tr>
<tr>
<td>-debug</td>
<td>prints out scenario-specific information useful for debugging</td>
</tr>
<tr>
<td>-limit</td>
<td>Change the number of cycles the game runs for. Default is 10,000,000. Set to 0 for unlimited cycles.</td>
</tr>
<tr>
<td>-randommap</td>
<td>Randomly generate scenario map with the current time as the seed. Potentially affects bot start position, scenario specific positions, general randomness. Note that this overrides -mapseed.</td>
</tr>
<tr>
<td>-mapseed &lt;int&gt;</td>
<td>Randomly generate scenario map based on the given seed. Seed should be a non-negative integer. Potentially affects bot start position, scenario specific positions, general randomness. Note that this overrides -randommap.</td>
</tr>
<tr>
<td>-randompuzzle</td>
<td>Randomly generate puzzles with the current time as the seed. Note that this overrides -puzzleseed.</td>
</tr>
<tr>
<td>-puzzleseed &lt;int&gt;</td>
<td>Randomly generate puzzles based on the given seed. Seed should be a non-negative integer. Note that this overrides -randompuzzle.</td>
</tr>
<tr>
<td>-drawcycles num</td>
<td>Causes the map to be redrawn every num cycles. The default is 8192, and lower values slow execution down, allowing movement to be observed much better</td>
</tr>
<tr>
<td>-largemap</td>
<td>Draws a larger map (but runs a little slower)</td>
</tr>
<tr>
<td>-maponly</td>
<td>Doesn't pop up the QtSpim window. Most useful when combined with -run.</td>
</tr>
<tr>
<td>-run</td>
<td>Immediately begins the execution of SPIMbot's program</td>
</tr>
<tr>
<td>-tournament</td>
<td>A command that disables the console, SPIM syscalls, and some other features of SPIM for the purpose of running a smooth tournament</td>
</tr>
<tr>
<td>-prof_file</td>
<td>specifies a file name to put gcov style execution counts for each statement. Make sure to stop the simulation before exiting, otherwise the file won’t be generated</td>
</tr>
<tr>
<td>-exit_when_done</td>
<td>Automatically closes SPIMbot when contest is over</td>
</tr>
<tr>
<td>-quiet</td>
<td>Suppress extraneous error messages and warnings</td>
</tr>
</tbody>
</table>

Note that -randommap and -mapseed override one another, and that -randompuzzle and -puzzleseed override one another. The flag that is typed last will be the overriding flag.

## Fun Command Line Arguments

Try running QtSpimbot with the -spaceship and -largemap flags!

## SPIMBot Physics

You don't need to understand these 100% to do well in SPIMBot, but they are provided in case anyone wants to use them to develop better SPIMBot code.
Position and Velocity

In the SPIM Universe, positions are given in pixels, or px.

Pixels start in the upper left at (x=0, y=0) and end in the bottom right at (x=300, y=300).

Velocities are given in pixels per 10000 cycles, or mips.

SPIMBot has a maximum speed of 10 mips. However, SPIMBot's engine can only provide 5 mips of directed velocity; the remaining 5 are provided by gravity, and always point left (the -x direction).

Gravity and Energy

SPIMBot is maneuvering near a star! In the SPIM Galaxy, stars have gravity, but it works differently than it does in ours. The local star's gravity is felt as a velocity, added to SPIMBot's engine velocity.

The star's velocity-effect is equal to:

\[ v = v_x + v_g = v_x + \left( -5 + \frac{x}{60}, 0 \right) \]

Intuitively, this represents a gradual increase in gravity from right to left.

So, if we set SPIMBot's velocity to \( v_g \) using the VELOCITY, ANGLE, and ANGLE_CONTROL IOs, the real velocity of SPIMBot will be:

\[ v = v_x + v_g = v_x + \left( -5 + \frac{x}{60}, 0 \right) \]

Now, this allows SPIMBot to go almost anywhere it wants, but at a price: the velocity created by SPIMBot's engines costs energy every cycle. In Lab 10, you will have unlimited energy; in LabSpimbot, you will not. This means that going close to the start costs more energy and takes longer than just staying near the station. However, the asteroids closer to the star are worth much more.

The energy cost per cycle is propotional to the SPIMbot velocity, with a constant factor of 1e-4.

Miscellaneous

- The boundary line where SPIMBot burns up in the star is \( x = 40 \).
- The space stations trace perfect circles at constant orbital velocity.
- Asteroids are always collectible.
- Under the hood, SPIMBot uses reasonably accurate floating point computations to update velocity and position; this means that you can use mathematical tools like integration to help plan your paths if you want to.

Source Code

TBD