“Bonk... Bonk... Bonk...” – A Poor SPIMBot

Contents

0 Document Updates 3

1 The Game 4

1.1 The Map 4

1.2 Tile Types 4

1.3 The Objective 5

1.4 Minibots 6

1.5 Silos 7

1.6 Scoring 7

2 SPIMBot I/O 9

2.1 Orientation Control 9

2.2 Odometry 9

2.3 Bonk (Wall Collision) Sensor 9

2.4 Timer 9

2.5 Puzzle 10

2.5.1 Dominosa 10

2.5.2 Puzzle Struct 10

2.5.3 Requesting a Puzzle 11

2.5.4 Submitting Your Solution 11

2.5.5 The Slow Solver 11

2.5.6 Debugging 12

3 Interrupts 13

3.1 Interrupt Acknowledgment 13
0 Document Updates

(Updates are listed in chronological order, newest updates are shown in red)

1. A reference in Section 6.1 incorrectly pointed to Section 5. It now correctly points to Section 1.6
2. Added more useful info to Sections 2.5.2 and 2.5.4
3. Created Section 2.5.6
4. Fixed the table in Appendix D
5. Removed redundant Appendix C
6. Updated/corrected Section 6.1
7. Added a new Appendix D
8. Binary has been upgraded to version 2020–11–25. This addresses the following issues:
   • The x and y coordinates of the minibot spawn position were flipped
   • Minibots would go into a wall when spawned
9. We have clarified which tile is accessed when doing \texttt{array[a][b]} in both \texttt{GET\_MAP} and \texttt{GET\_KERNEL\_LOCATIONS}
10. \texttt{ARENA\_MAP} is deprecated and so was removed from Appendix A
11. Binary has been upgraded to version 2020–12–2. This addresses major bugs regarding silos.  
    If you are experiencing any issues regarding silos, \textit{please try downloading this update!}
12. Corrected the board size from 32 × 32 tiles to 40 × 40 tiles
1 The Game

For this year’s SPIMBot competition, you will be writing code to control a bot to collect corn from the Morrow Plots, deploy minibots, and build silos.

1.1 The Map

The SPIMBot map looks like this:

SPIMBot 1 (red) always starts in the top left corner (Tile [0, 0], pixels [4, 4]). SPIMBot 2 (blue) always starts in the bottom right corner (Tile [39, 39], pixels [316, 316]).

Note: During grading and tournament qualification, your SPIMBot will always be bot #1.

The map is rendered on a 320×320 pixel board, so each tile is 8 × 8 pixels.

1.2 Tile Types

Below is a summary of the different tile types:
<table>
<thead>
<tr>
<th>Tile type</th>
<th>Image</th>
<th>Tile Code</th>
<th>Can Collide with Player</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floor</td>
<td><img src="image1.png" alt="Floor Image" /></td>
<td>0</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Wall</td>
<td><img src="image2.png" alt="Wall Image" /></td>
<td>1</td>
<td>Yes</td>
<td>There is a mirrored version of this as well. It represents the exact same tile as the original</td>
</tr>
<tr>
<td>Silo</td>
<td><img src="image3.png" alt="Silo Image" /></td>
<td>2 or 3 (see Notes!)</td>
<td>No</td>
<td>Tiles that have your silo have a code of 2. Tiles that have your opponent’s silo will have a code of 3.</td>
</tr>
</tbody>
</table>

**Note:** The wall sprites appear more than 1 tile tall, but their collision box in game is 1 square tile.

You can access the map using the `GET_MAP` command. It’ll return a struct that contains a 2D array with each entry corresponding to a tile on the map. The value of each entry will be the tile code of the corresponding tile. See Appendix A and Appendix B for more information!

### 1.3 The Objective

Your goal is simple: collect as many kernels of corn as possible! Scattered across the map will be corn that appears as yellow dots or as a full ear. Their appearance varies based on the number of kernels they contain:

- ![1 kernel](image4.png)
- ![2 kernels](image5.png)
- ![3 kernels](image6.png)
- ![4-9 kernels](image7.png)
- ![10+ kernels](image8.png)

To find out how many kernels are at any tile, use the `GET_KERNEL_LOCATIONS` command. Random amounts of kernels will periodically spawn at random locations on the map, so don’t worry about running out of corn any time soon! Just make sure you grab them faster than your opponent does...

You can collect kernels with your SPIMBot by moving it to a tile that has kernels on it and then using the `PICKUP` command. Kernels collected with your SPIMBot count towards your score.

You can also collect kernels with any minibots you spawn. However, kernels collected with minibots...
To have the kernels collected with the minibots count towards your score, you need to deposit the kernels into a silo. All corn deposited into a silo is immediately transferred to the SPIMBot. You can build a silo if you have enough kernels and minibots.

For more information about scoring, see Section 1.6.

Another important aspect of the game is solving puzzles. Solving a puzzle gets you 1 puzzlecoin. Puzzlecoins don’t contribute to your score. They’re needed (along with kernels) to spawn minibots. More info about the puzzles can be found in Section 2.5.

1.4 Minibots

If you have the required amount of kernels and puzzlecoins, you can spawn minibots that will help you gather kernels and potentially destroy your opponent’s silos. There are two type of minibots to choose from:

1. Basic minibots – These minibots will simply keep moving forward. The start off by going in a random direction, and when they hit a wall they will bounce according to the simple rule of reflection: angle in equals angle out (with just a little bit randomness thrown in so that the minibot never gets stuck). Note that you cannot give these bots a target tile or even an initial direction to head out in!

   Spawning a basic minbot requires 1 puzzlecoin and 2 kernels.

2. Advanced minibots – This type of minbot is able to move to a specific target tile. To use this type of minbot, you first need to select which advanced minibots you would like to send out by one of the commands that select minibots (see Appendix A for a list of them). Once you’ve selected some of the advanced minibots, you set their target tile using the SET_TARGET_TILE command. The advanced minibots will then head towards their target tile using a pathfinding algorithm. The details are boring, but be assured that the advanced minibots will find a pretty efficient way to get to their target! Once they arrive, they will stop and stay at their target tile until given a new target.

   Spawning an advanced minbot requires 2 puzzlecoins and 2 kernels.

Both types of minibots will pick up kernels automatically. That is, if they move over a tile that has kernels on it, they will pick up the corn with no command needed. Remember, this isn’t true for the SPIMBot; you need to move it to the kernels and then also use the PICKUP command.

If you have the proper requirements, then you can spawn a minibot (whose type is of your choosing) using the SPAWN_MINIBOT command. This will spawn a minibot at the same location as your SPIMBot. When it comes to satisfying the requirements, only the kernels that your SPIMBot has will count! After spawning the minibots, the number of kernels and puzzlecoins you have will reduce by the amount of each you spent. Your score, however, will remain the same.

Minibots of different colors will annihilate each other in an explosion. On top of that, all minibots in a 3 × 3 region centered at the tile where the collision occurred will also be destroyed. When a minibot is destroyed, it drops any kernels it contained on the tile where it was destroyed. Note that silos are not destroyed in the blast.
Minibots and SPIMBots (each being of any color) do not interact with each other. Minibots of the same color will also not interact with each other.

You can get info about your minibots by using the GET_MINIBOT_INFO command. You can see info such as how many bots you have, where they are, how many kernels they’ve collected, etc. You can get the exact same information about your opponent’s bots too; simply use the GET_OPPONENT_MINIBOT_INFO command instead. See Appendix A and Appendix B for more info.

1.5 Silos

You need to build silos so that you can transfer corn from your minibots to your SPIMBot. Only the corn that your SPIMBot has contributes to your score and can be used for spawning minibots or building more silos.

To build a silo, you need to have 10 kernels and you need to send 3 minibots to the tile you want to build a silo on. Once these conditions are met, you can use the BUILD_SILO command. Any corn that the sacrificed minibots had will be dropped onto the tile they were sacrificed on, i.e. the tile that the silo is now on. Bots can move over silo tiles, so the kernels can be picked back up by you.

To have a minibot deposit its corn into a silo, it simply needs to go to the tile that a silo is on (this is much easier to do with an advanced minibot!). Once the minibot reaches the silo, the corn is instantly transferred to your SPIMBot.

You are able to destroy your opponent’s silos. If one of your minibots is on the same tile as one of your opponent’s silos, and if your opponent has no minibots on that tile, then the silo will be destroyed. Since the silos don’t actually store corn, they don’t drop anything when they are destroyed. However, remember that a minibot will drop its kernels when destroyed.

If one of your minibots is on the same tile as one of your opponent’s silos but your opponent does have a minibot on the same tile, then the two minibots will annihilate each other and destroy all minibots in a 3 × 3 region centered at the silo. The silo itself will remain unharmed though. Again, remember that minibots will drop their kernels when destroyed.

Note: Your opponent can attack your silos as well, and so you might have to defend them!

To get the locations of your and your opponent’s silos, use the GET_MAP command. If a tile has a tile code of 2 then your silo is at that tile. If a tile has a tile code of 3, your opponent’s silo is at that tile. See Appendix A and Appendix B for more info.

1.6 Scoring

For the tournament, you get 1 point for every kernel your SPIMBot collects. You also get 1 point for every kernel that your minibots transfer to your SPIMBot via silos.

Even if you spend kernels to spawn minibots or build silos, your score won’t go down!

Whoever has a higher score at the end of a round wins the round. In the case of a tie, a winner is chosen randomly.
There’s two important things to remember:

1. PUZZLES DON’T COUNT TOWARDS YOUR FINAL SCORE!!

2. If your minibots have collected kernels, you need to transfer them to your SPIMBot via silos if you want them to count towards your score.

⚠️ Due to our incomplete balance testing, the exact scoring numbers may change before the final SPIMBot competition, though this is unlikely. ⚠️
2 SPIMBot I/O

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 - 0xffffffff. Below we describe SPIMBot’s I/O devices in more detail.

A comprehensive list of all the I/O addresses can be found in the Appendix.

2.1 Orientation Control

SPIMBot’s orientation can be controlled in two ways:

1. By specifying an adjustment relative to the current orientation
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 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 the SPIMBot clockwise. While it may not sound intuitive, this matches the normal Cartesian coordinates (the +x direction is 0°, +y is 90°, −x is 180°, and −y is 270°), since we consider the top-left corner to be (0,0) with +x and +y being right and down, respectively. For more details see section SPIMBot Physics.

2.2 Odometry

Your SPIMBot has sensors that tell you its current position. Reading from addresses BOT_X (0xffff0020) and BOT_Y (0xffff0024) will return the x-coordinate and y-coordinate of your SPIMBot respectively, in pixels. Storing to these addresses, unfortunately, does nothing. (You can’t teleport.)

2.3 Bonk (Wall Collision) Sensor

The bonk sensor signals an interrupt whenever SPIMBot runs into a wall.

Note: Your SPIMBot’s velocity is set to zero when it hits a wall.

2.4 Timer

The timer does two things:

1. The number of cycles elapsed since the start of the game can be read from TIMER (0xffff001c).
2. A timer interrupt can be requested by writing the cycle number at which the interrupt is desired to the TIMER. It is very useful for task-switching between solving puzzles and
moving.

2.5 Puzzle

Solving puzzles is the only way to earn puzzlecoins. The puzzle you have to solve is Dominosa. It is similar to Lab 7, except we have modified the struct slightly. Your Lab 7 solver will not work without modification for specific edge cases! Use the provided solution instead!

2.5.1 Dominosa

Dominosa is a puzzle where you try to tile a board with dominos while making sure that no domino is used more than once. This is a deceptively simple concept and like most puzzle games, it can be shown to be NP-hard (see https://cs.stackexchange.com/questions/16850/is-dominosa-np-hard). You can find an example of the puzzle here: https://www.puzzle-dominosa.com/. Below is an example of a puzzle board.

![Example of a puzzle board](image)

2.5.2 Puzzle Struct

The puzzles come in a struct. The struct written to memory is defined as follows:

```c
#define MAX_GRIDSIZE 16
#define MAX_MAXDOTS 15
typedef struct Dominosa {
    int num_rows;
    int num_cols;
    int max_dots;
    unsigned char board[MAX_GRIDSIZE * MAX_GRIDSIZE];
    unsigned char dominos_used[MAX_MAXDOTS * MAX_MAXDOTS + 1];
} dominosa_question;
```

The number contained in row $y$ and column $x$ in the Dominosa puzzle will be found at `board[y * num_cols + x]`. The contents of `dominos_used` are initially set to 0. This space is used by the slow solver to help compute the solution. If you are writing your own solver, you can use this space however you like.

A list of possible puzzle dimensions can be found in Appendix C.
2.5.3 Requesting a Puzzle

The first step to getting a puzzle is to allocate space for it in the data segment, then write a pointer of that space to REQUEST_PUZZLE (0xffff00d0). However, it takes some time to generate a puzzle, so you will have to wait for a REQUEST_PUZZLE interrupt. When you get the REQUEST_PUZZLE interrupt, the puzzle struct for you to solve will be in the allocated space with the address you gave. Note that you must enable the REQUEST_PUZZLE interrupt or else you will never receive a puzzle.

To accept puzzle interrupts, you must turn on the mask bit specified by REQUEST_PUZZLE_INT_MASK (0x800). You must acknowledge the interrupt by writing a nonzero value to REQUEST_PUZZLE_ACK (0xffff00d8). The puzzle will then be stored in the pointer written to REQUEST_PUZZLE.

You can request more puzzles before solving the previous ones. But be sure to submit the solution in the same order as you requested them.

2.5.4 Submitting Your Solution

After solving the puzzle, you need to submit the solution to earn puzzlecoins. To submit your solution, simply write a pointer of the your solution board to SUBMIT_SOLUTION. If your solution is correct, you will be rewarded with **1 puzzlecoin**!

The requirements for the submitted solution are:

1. The solution should be an unsigned char array with at least num_rows * num_cols elements.
2. The element at the y * num_cols + x position will be used to find a domino for the number at row y column x on the board.
3. The same domino should use the same nonzero number in the solution board.
4. Different dominos adjacent to each other should use different numbers.

Run with the -debug flag, request and fail a puzzle to see examples of potential solutions.

2.5.5 The Slow Solver

We’ve given you a slow solver that you can use in your SpimBot. You can find it in _release along with the starter code. The slow solver uses a recursive backtracking algorithm. It tries all possible combinations of dominos and positions. This can be greatly sped up and it is a good idea to work on this first after you have built your SPIMBot.

```c
// the slow solve entry function,
// solution will appear in solution array
// return value shows if the dominosa is solved or not
int slow_solve_dominosa(dominosa_question* puzzle, unsigned char* solution) {
    // zeroes the array
    zero(puzzle->num_rows * puzzle->num_cols, solution);
    zero(sizeof(puzzle->dominos_used), puzzle->dominos_used);
    return solve(puzzle, solution, 0, 0);
}
```
The arguments are the same ones as the solver given in Lab7.

If you wish to use the slow solver, you will need to allocate some space in the data segment for the solution to be stored. Then, pass the address as the second argument.

You may look at file “p2_main.s” in Lab 7 to learn how to use the slow solver.

2.5.6 Debugging

If you are writing your own solver, run your solution with the –debug flag for some useful information. When a board or solution is printed out, the (0,0) position will be the top left corner. If your solution failed, it will tell you if a domino is used twice, a number in your solution array is 0, or if it cannot determine an ambiguous domino.

In the case of an ambiguous domino, a position and an error code will be given. The error code ranges from 0 to 15. Each bit in the error code represents a direction that the number in the given position might be matched to given your solution. 1 is up, 2 is left, 4 is down, 8 is right.

For example, if you submitted the solution 1 1 1 2 on a 1 × 4 board, the error "Puzzle not solved! Ambiguous domino at (0,1), with error code 10" will be given because we cannot determine if a domino should be placed on the left or in the middle. I.e. the number at position (0, 1) might be match to the left or to the right.
3 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>

3.1 Interrupt Acknowledgment

When 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 any value to a memory-mapped I/O location.

In all cases, writing the acknowledgment addresses with any value will clear 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>0xffffffff06c</td>
</tr>
<tr>
<td>Bonk (wall collision)</td>
<td>0x1000</td>
<td>0xffffffff060</td>
</tr>
<tr>
<td>Request Puzzle</td>
<td>0x0800</td>
<td>0xffffffff00d8</td>
</tr>
</tbody>
</table>

3.2 Bonk

You will receive the **Bonk** interrupt if your SPIMBot runs into a wall. Your SPIMBot’s velocity will also be set to zero if it runs into a wall.

3.3 Request Puzzle

You will receive the **Request Puzzle** interrupt once the requested puzzle is ready to be written into the provided memory address. You must acknowledge this interrupt for the puzzle to be written to memory!
4 SPIMBot Physics

4.1 Position and Velocity

In the SPIM Universe, positions are given in pixels. Pixels start in the upper left at $(x = 0, y = 0)$ and end in the bottom right at $(x = 320, y = 320)$. Just as with the Cartesian plane, the x-coordinate increases as you go to the right. However, unlike the Cartesian plane, the y-coordinate increases as you go down. (This is a common convention in graphics programming)

An angle of $0^\circ$ is parallel to the positive x-axis. As the angle moves clockwise, it increases. When the angle is parallel to the positive y-axis, it’s at $90^\circ$.

![Diagram of angles and coordinates](image)

The position of the SPIMBot is where the coordinates of its center. The SPIMBot itself is just a circle with a radius of 3 pixels, centered around the SPIMbot position.

SPIMBot velocity is measured in units of pixels/10,000 cycles. This means that at maximum speed ($\pm 10$), the SPIMBot moves at a speed of 1 pixel per 1000 cycles, or 1 tile (8 pixels) per 8000 cycles.

The SPIMBot has no angular nor linear acceleration. This means that you can rotate the SPIMBot instantly by using the `ANGLE` and `ANGLE_CONTROL` commands. See Section 2.1 for more details.

4.2 Collisions

If your position is about to go out-of-bounds (either less 0 or greater than 320 on either axis) or cross into an impassible cell, your velocity will be set to zero and you will receive a bonk interrupt.

**Note:** Your position is the center of your SPIMBot! This means that your SPIMBot will partially overlap the wall before it “collides” with it.
5 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` and `-file2` arguments. 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, although it can modify timings and change the behavior of the game. 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`

Note: the `-tournament` flag will suppress MIPS error messages!

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 breeze` to your command line arguments.

5.1 Useful Command Line Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-file &lt;file1.s&gt; &lt;file2.s&gt; ...</code></td>
<td>Specifies the assembly file(s) to use</td>
</tr>
<tr>
<td><code>-file2 &lt;file1.s&gt; &lt;file2.s&gt; ...</code></td>
<td>Specifies the assembly file(s) to use for a second SPIMBot</td>
</tr>
<tr>
<td><code>-part1</code></td>
<td>Run SPIMBot under Lab 9 part 1 conditions</td>
</tr>
<tr>
<td><code>-part2</code></td>
<td>Run SPIMBot under Lab 9 part 2 conditions</td>
</tr>
<tr>
<td><code>-test</code></td>
<td>Run SPIMBot starting with 65535 money. Useful for testing</td>
</tr>
<tr>
<td><code>-debug</code></td>
<td>Prints out scenario-specific information useful for debugging</td>
</tr>
<tr>
<td><code>-limit</code></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><code>-randommap</code></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 <code>-mapseed</code></td>
</tr>
<tr>
<td><code>-mapseed &lt;seed&gt;</code></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 <code>-randommap</code></td>
</tr>
<tr>
<td>Argument</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>-randompuzzle</code></td>
<td>Randomly generate puzzles with the current time as the seed. Note that this overrides <code>-puzzleseed</code></td>
</tr>
<tr>
<td><code>-puzzleseed &lt;seed&gt;</code></td>
<td>Randomly generate puzzles based on the given seed. Seed should be a non-negative integer. Note that this overrides <code>-randompuzzle</code></td>
</tr>
<tr>
<td><code>-drawcycles &lt;num&gt;</code></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><code>-largemap</code></td>
<td>Draws a larger map (but runs a little slower)</td>
</tr>
<tr>
<td><code>-smallmap</code></td>
<td>Draws a smaller map (but runs a little faster)</td>
</tr>
<tr>
<td><code>-maponly</code></td>
<td>Doesn’t pop up the QtSpim window. Most useful when combined with <code>-run</code></td>
</tr>
<tr>
<td><code>-run</code></td>
<td>Immediately begins the execution of SPIMBot’s program</td>
</tr>
<tr>
<td><code>-tournament</code></td>
<td>A command that disables the console, SPIM syscalls, and some other features of SPIM for the purpose of running a smooth tournament. Also forces the map and puzzle seeds to be random. This includes disabling error, which can make debugging more difficult</td>
</tr>
<tr>
<td><code>-prof_file &lt;file&gt;</code></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><code>-exit_when_done</code></td>
<td>Automatically closes SPIMBot when contest is over</td>
</tr>
<tr>
<td><code>-quiet</code></td>
<td>Suppress extraneous error messages and warnings</td>
</tr>
<tr>
<td><code>--version</code></td>
<td>Prints the version of the binary (note the double-dash!)</td>
</tr>
</tbody>
</table>

**Latest version:** 2020-12-2

**Tip:** If you’re trying to optimize your code, run with `-prof_file <file>` to dump execution counts to a file to figure out which areas of your code are being executed more frequently and could be optimized for more benefit!

Note that `-randommap` and `-mapseed` override one another, and that `-randompuzzle` and `-puzzleseed` override one another. The `-tournament` flag also overrides most other flags. The flag that is typed last will be the overriding flag.
6 Tournament Rules

6.1 Qualifying Round

To qualify for the SPIMBot tournament, you need to collect 100 kernels in at least 3 games out of a total 4 games. Each of these 4 games will use a different map seed. **Note that the map seed controls the placement of the kernels, not the placement of the walls!** This is the command we will use to run your code on some seed X for qualifications:

```
QtSpimbot -f spimbot.s -mapseed [mapseed]
```

For more details on how your bot will be scored in these games, see Section 1.6.

6.2 Tournament Rounds

Once you have qualified, you will then have to compete in a tournament against your classmates. For the tournament rounds, your bot will be randomly paired with another bot. The bot that have the most score at the end of the round will win. In the case of a tie, the winner will be selected at random. The tournament might be round-robin, double-elimination, etc., depending on the number of people qualified.

We will use the following command to run two different spimbots against each other:

```
QtSpimbot -file spimbotA.s -file2 spimbotB.s -tournament
```

**NOTE:** Your bot may spawn in either the upper left or lower right corner!

6.3 LabSpimbot Grading

LabSpimbot grade breakdown is specified in the LabSpimbot handout.
### Appendix A: MMIO Commands

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Acceptable Values</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>VELOCITY</td>
<td>0xffff0010</td>
<td>-10 to 10</td>
<td>Current velocity</td>
<td>Updates velocity of the SPIMBot</td>
</tr>
<tr>
<td>ANGLE</td>
<td>0xffff0014</td>
<td>-360 to 360</td>
<td>Current orientation</td>
<td>Updates angle of SPIMBot when <strong>ANGLE_CONTROL</strong> is written</td>
</tr>
<tr>
<td>ANGLE_CONTROL</td>
<td>0xffff0018</td>
<td>0 (relative) 1 (absolute)</td>
<td>N/A</td>
<td>Updates angle to last value written to <strong>ANGLE</strong></td>
</tr>
<tr>
<td>TIMER</td>
<td>0xffff001c</td>
<td>Anything</td>
<td>Number of elapsed cycles</td>
<td>Timer interrupt when elapsed cycles == write value</td>
</tr>
<tr>
<td>TIMER_ACK</td>
<td>0xffff006c</td>
<td>Anything</td>
<td>N/A</td>
<td>Acknowledge timer interrupt</td>
</tr>
<tr>
<td>BONK_ACK</td>
<td>0xffff0060</td>
<td>Anything</td>
<td>N/A</td>
<td>Acknowledge bonk interrupt</td>
</tr>
<tr>
<td>REQUEST_PUZZLE_ACK</td>
<td>0xffff00d8</td>
<td>Anything</td>
<td>N/A</td>
<td>Acknowledge request puzzle interrupt</td>
</tr>
<tr>
<td>RESPAWN_ACK</td>
<td>0xffff00f0</td>
<td>Anything</td>
<td>N/A</td>
<td>Acknowledge respawn interrupt</td>
</tr>
<tr>
<td>BOT_X</td>
<td>0xffff0020</td>
<td>N/A</td>
<td>N/A</td>
<td>Current X-coordinate, px</td>
</tr>
<tr>
<td>BOT_Y</td>
<td>0xffff0024</td>
<td>N/A</td>
<td>N/A</td>
<td>Current Y-coordinate, px</td>
</tr>
<tr>
<td>SCORES_REQUEST</td>
<td>0xffff1018</td>
<td>Valid data address</td>
<td>N/A</td>
<td>M[address] = [your score, opponent score]</td>
</tr>
<tr>
<td>REQUEST_PUZZLE</td>
<td>0xffff00d0</td>
<td>Valid data address</td>
<td>N/A</td>
<td>M[address] = new puzzle; sends Request Puzzle interrupt when ready</td>
</tr>
<tr>
<td>Instruction</td>
<td>Address</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SUBMIT_SOLUTION</strong></td>
<td>0xffff00d4</td>
<td>Valid data address</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Submits puzzle solution at M[address]</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>PICKUP</strong></td>
<td>0xffff00f4</td>
<td>Anything</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Picks up any kernels on the SPIMBot’s current tile</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SPAWN_MINIBOT</strong></td>
<td>0xffff00dc</td>
<td>0 (basic minibot) 1 (advanced minibot)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Spawns specified type of minibot at SPIMBot location</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SELECT_MINIBOTS_IN_REGION</strong></td>
<td>0xffff00e0</td>
<td>Four 1-byte coordinates concatenated: $[x_1, y_1, x_2, y_2]$. $y_2$ is the LSB</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Selects all advanced minibots in the rectangle whose top-left corner is tile $(x_1, y_1)$ and whose bottom-right corner is $(x_2, y_2)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SELECT_MINIBOT_BY_ID</strong></td>
<td>0xffff2004</td>
<td>The ID of the advanced minibot you want to select</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Selects the advanced minibot that has the given ID. Note that this selects only <em>one</em> advanced minibot.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SELECT_IDLE_MINIBOTS</strong></td>
<td>0xffff00e4</td>
<td>Anything</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Selects all the advanced minibots that haven’t been ordered to move or have reached their target tile</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SET_TARGET_TILE</strong></td>
<td>0xffff00e8</td>
<td>Two 1-byte coordinates concatenated: $[x_{\text{target}}, y_{\text{target}}]$. $y_{\text{target}}$ is the LSB.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sets the target tile of the <em>currently selected</em> advanced minibots to $(x_{\text{target}}, y_{\text{target}})$</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>BUILD_SILO</strong></td>
<td>0xffff0000</td>
<td>Two 1-byte coordinates concatenated: $[x_{\text{silo}}, y_{\text{silo}}]$. $y_{\text{silo}}$ is the LSB.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Build silo at tile $(x_{\text{silo}}, y_{\text{silo}})$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Command</td>
<td>Address</td>
<td>Data Type</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------</td>
<td>--------------------</td>
<td>-------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>GET_MAP</td>
<td>0xffff00f0</td>
<td>Valid data address</td>
<td>Writes Map struct to given memory location</td>
<td></td>
</tr>
<tr>
<td>GET_PUZZLE_CNT</td>
<td>0xffff2008</td>
<td>Valid data address</td>
<td>Writes Num_Puzzles struct to given memory location</td>
<td></td>
</tr>
<tr>
<td>GET_KERNEL_LOCATIONS</td>
<td>0xffff200c</td>
<td>Valid data address</td>
<td>Writes Kernel_Locations struct to given memory location</td>
<td></td>
</tr>
<tr>
<td>GET_NUM_KERNELS</td>
<td>0xffff2010</td>
<td>Valid data address</td>
<td>Writes Num_Kernels struct to given memory location</td>
<td></td>
</tr>
<tr>
<td>GET_MINIBOT_INFO</td>
<td>0xffff2014</td>
<td>Valid data address</td>
<td>Writes your MiniBot_Info struct to given memory location</td>
<td></td>
</tr>
<tr>
<td>GET_OPPONENT_MINIBOT_INFO</td>
<td>0xffff2018</td>
<td>Valid data address</td>
<td>Writes opponent’s MiniBot_Info struct to given memory location</td>
<td></td>
</tr>
</tbody>
</table>

1. If you don’t meet the necessary requirements, this command will do nothing
2. These commands have no effect on a basic minibot
Appendix B: Struct Definitions

```c
struct Map {
    char map[40][40]; // map[a][b] = 0 if tile (x,y) is plain ground
    // map[a][b] = 1 if tile (x,y) is a wall
    // map[a][b] = 2 if tile (x,y) has YOUR silo
    // map[a][b] = 3 if tile (x,y) has your OPPONENT'S silo
}

struct Num_Puzzles {
    int player_puzzles; // Number of puzzles YOU have solved
    int opponent_puzzles; // Number of puzzles your OPPONENT has solved
}

struct Num_Kernels {
    int player_kernels; // Number of kernels YOU have
    int opponent_kernels; // Number of kernels your OPPONENT has
}

struct Kernel_Locations {
    int num_kernel_locations; // Number of tiles that have 1+ kernels
    char kernels[40][40]; // kernels[a][b] = number of kernels at
                           // tile (b,a) ~NOTE THE DIFFERENCE!!~
}

struct MiniBot {
    int ID; // ID of the minibot. This is unique!
    char type; // 0 = basic, 1 = advanced
    char x; // The TILE x coordinate of the bot
    char y; // The TILE y coordinate of the bot
    char kernels; // Number of kernels the bot has collected
}

struct MiniBot_Info {
    int num_minibots; // Number of minibots Player X has
    struct MiniBot minibots[num_bots]; // Info about Player X's minibots
                                           // (see MiniBot struct definition)
}
```

// NOTE: if you used the GET_MINIBOT_INFO command to get this struct, then
// Player X is YOU. If you used the GET_OPPONENT_MINIBOT_INFO command to get
// this struct, then Player X is YOUR OPPONENT.
Appendix C: Puzzle Dimensions

The parameters of each Dominosa puzzle are selected from the following list using a simple weighted distribution.

<table>
<thead>
<tr>
<th>num_rows</th>
<th>num_cols</th>
<th>max_dots</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>7</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>
Appendix D: Helpful Code

.data
three: .float 3.0
five: .float 5.0
PI: .float 3.141592
F180: .float 180.0

.text
# -----------------------------------------------------------------------
# sb_arctan - computes the arctangent of y / x
# $a0 - x
# $a1 - y
# returns the arctangent
# -----------------------------------------------------------------------
.globl sb_arctan
sb_arctan:
    li $v0, 0 # angle = 0;
    abs $t0, $a0 # get absolute values
    abs $t1, $a1
    ble $t1, $t0, no_TURN_90
    ## if (abs(y) > abs(x)) { rotate 90 degrees }
    move $t0, $a1 # int temp = y;
    neg $a1, $a0 # y = -x;
    move $a0, $t0 # x = temp;
    li $v0, 90 # angle = 90;
no_TURN_90:
    bgez $a0, pos_x # skip if (x >= 0)
    ## if (x < 0)
    add $v0, $v0, 180 # angle += 180;
pos_x:
    mtc1 $a0, $f0
    mtc1 $a1, $f1
    cvt.s.w $f0, $f0 # convert from ints to floats
    cvt.s.w $f1, $f1
div.s $f0, $f1, $f0 # float v = (float) y / (float) x;
mul.s $f1, $f0, $f0 # v^^2
mul.s $f2, $f1, $f0 # v^^3
l.s $f3, three # load 3.0
div.s $f3, $f2, $f3 # v^^3/3
sub.s $f6, $f0, $f3 # v - v^^3/3
mul.s $f4, $f1, $f2 # v^^5
l.s $f5, five # load 5.0
div.s $f5, $f4, $f5 # v^^5/5
add.s $f6, $f6, $f5 # value = v - v^^3/3 + v^^5/5
l.s $f8, PI # load PI
div.s $f6, $f6, $f8 # value / PI
l.s $f7, F180 # load 180.0
mul.s $f6, $f6, $f7 # 180.0 * value / PI
cvt.w.s $f6, $f6 # convert "delta" back to integer
mfc1 $t0, $f6
add $v0, $v0, $t0 # angle += delta
jr $ra
euclidean_dist:
    mul $a0, $a0, $a0  # x^2
    mul $a1, $a1, $a1  # y^2
    add $v0, $a0, $a1  # x^2 + y^2
    mtc1 $v0, $f0
    cvt.s.w $f0, $f0  # float(x^2 + y^2)
    sqrt.s $f0, $f0    # sqrt(x^2 + y^2)
    cvt.w.s $f0, $f0   # int(sqrt(...))
    mfc1 $v0, $f0
    jr $ra