"Bonk. . . Bonk. . . Bonk. . ." – A Poor SPIMBot

Contents

0 Document Updates

Updates are listed in chronological order, newest updates are shown in red. We will always attempt to fix game-breaking bugs. Other bugs that get reported may get a new release or might just get a documentation change.

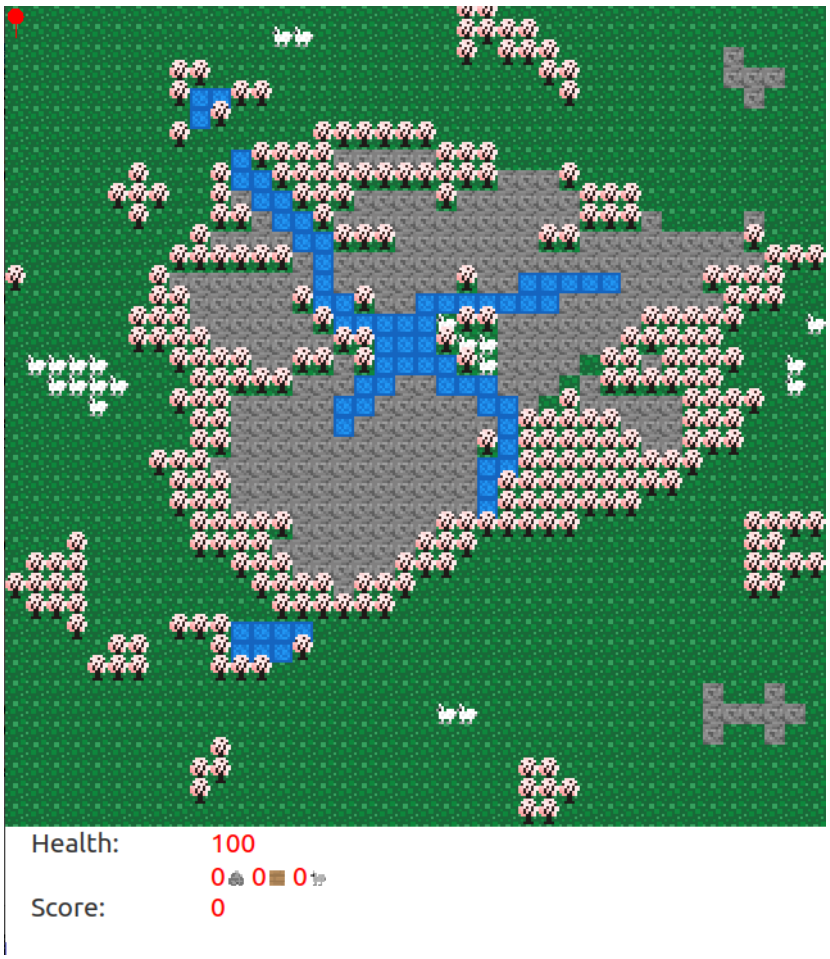
1. Fixed info about Chests in Section 1.7, changed mentions of "bears" to "squirrels"
2. Added a note that corners are not necessary for a valid base.
3. Patched an bug in the code that allowed a bot to craft without creativity
4. Fixed an issue with invalid overflows being printed out to the console
5. Added information on return codes for SUBMIT_SOLUTION
6. Added note about GET_PUZZLE_CNT not working
7. The submitter must have created and pushed a file called teamname.txt in their Lab14 folder. This file should contain your team's name in 25 plain ASCII characters.

1 The Game

For this year's SPIMBot competition, you will be writing code to control a bot to survive on on a remote island. Your bot will collect resources and craft items in order to survive.

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]). Indicators below the map display health, thirst, inventory, and score. Health can range from 0 to 100, and a water block will appear to the right of your bot's health if it is thristy. We show a limited view of the inventory below the health, representing the number of stone, wood, and wool in the bot's inventory. Finally, the score, which can be positive or negative, is depicted at the bottom.

Note: During lab 10, grading, and tournament qualification, **your SPIMBot will always be bot #1**.

The map is rendered on a 320×320 pixel board with 40×40 tiles, so each tile is 8×8 pixels.

1.2 Tile Types

Below is a summary of the different tile types:

Tile type	Image	Tile Code	Can Collide with Player	Durability	Notes
Forest		0	No	6000	Required for every crafting recipe.
Stone		1	No	8000	Useful for crafting doors and stone walls.
Sheep		2	No	1500	Useful for crafting beds.
Wood Wall		3	Yes	100000	For building forts.
Stone Wall		4	Yes	200000	For building stronger forts.
Bed		5	No	2000	Set your bot's spawn point and lose less points when it dies.
Chest		6	No	4000	This is used to store items, and is a way to densely pack resources to increase a fort's value. Items inside a chest are lost when the chest is broken.
Door		7	No	100000	While players can pass through this, squirrels cannot.
Floor/Ground		8	No	N/A	
Water		9	Yes	N/A	Using this block allows you to drink

In addition to these options, it is important to note that tiles have an owner. This is mostly

important for beds because only the owner of a bed can respawn at that bed.

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 contain information about the corresponding tile, including the tile type, owner, and durability. See Appendix A and Appendix B for more information! Note that attempts to break a block take away from its durability.

1.3 Items & Fort Grading

Items in this game can be collected by breaking blocks or by crafting them using other items. Each item in the game has a value assigned to it, shown below. You may notice that a chest's value looks a little different: this is because a chest's value is equal to 7 plus the sum of the values of all the items it contains. If a chest has 3 wool inside, then the chest is worth $7 + 3 * 3 = 16$ points.

Tile type	**Tile Code**	**Value**
Wood	0	1
Stone	1	2
Wool	2	3
Wood Wall	3	3
Stone Wall	4	6
Bed	5	13
Chest	6	$7 + x$
Door	7	5
Ground	8	1

The item value is used for granting players points during the game. You can earn points by crafting an item: when you do so, you will earn points equal to the value of the item.

You can also earn points by building a base out of Wood Walls, Stone Walls, and Doors, then

submitting it for grading using the `SUBMIT_BASE` MMIO. To earn points for a fort you've built, you must submit a location that is fully enclosed by walls and/or doors. No other blocks will do! The value of a valid fort is calculated by adding up the values of all the beds, chests, and floor space within the walls of your fort. It is also important to know that the edge of the map does not count as part of the border. Also, corners are not mandatory for a valid base but they may be a good idea to protect yourself from squirrels. You can submit your base location at any time, but you will only be awarded points for your base once at the end of each night. Make sure your base doesn't get destroyed during the night!

1.4 Recipes

The following recipes exist in the game

$$
\begin{aligned}
WoodWall &\Leftarrow 2 * Wood \\
StoneWall &\Leftarrow Wood + 2 * Stone \\
Bed &\Leftarrow 3 * Wool + 3 * Wood \\
Chest &\Leftarrow 6 * Wood \\
Door &\Leftarrow 2 * Wood + Stone
\end{aligned}
$$

1.5 Inventory

An inventory is just a dictionary mapping items to the amount of that item it contains. Inventories have a cap on how many of each type of item they can hold. Two entities have inventories in the game: Bots and Chests. While Bots have a capacity of 8 of each item, Chests can store up to 16 of each.

1.6 The Objective

Your goal is simple: gather resources, craft them, and survive! Scattered across the map are blocks that yield useful resources when broken. In Lab 9, you played a stripped down singleplayer version of the game. Now you will need to stay hydrated and build a fort in order to protect yourself from squirrels and survive the night. Once you've got surviving figured out, you can also try to get in your opponent's way or loot their base.

Another important aspect of the game is solving puzzles. Solving a puzzle gets you 1 **creativity**. Creativity doesn't contribute to your score, but it's needed (along with resources) to craft items. More info about the puzzles can be found in Section 2.5.

1.7 The Spimbot

Your spimbot executes the code you write for it and uses MMIO to interact with the world. Your spimbot needs to be conscientious about its health and hydration. If your bot is dehydrated, it will begin to overheat and lose health. When your bot's hydration is 0, you lose 1 health every 10,000 cycles. You can rehydrate by using the MMIO `USE_BLOCK` on a water tile. If your health reaches 0, your bot dies. When your bot dies, it will respawn at your most recently placed bed, or if none is available, it will respawn at your start point.

Your player has a range of 3 tiles (calculated using euclidean distance between tiles). If you try to target a tile outside of this range, your MMIO call will not succeed. Additionally, your bot is strapped with an inventory. Your bot can hold up to 8 of each type of item in the game.

You bot can attack by using `ATTACK`. Please note that you are rate-limited on how fast you can perform attacks. Specifically, you cannot attack within 100 cycles of placing a block or attacking.

Your bot can place blocks by using `PLACE_BLOCK` if you have the block in your inventory and you are placing the block on ground.

Your bot can break blocks by using `BREAK_BLOCK`, this deals 10 damage to the durability of the block. If the block breaks, then you become the owner of the block and it is added into your inventory unless you do not have space for it.

Additionally, you can use blocks through `USE_BLOCK`. If you perform `USE_BLOCK` on a bed, you become the owner of the bed. If the item is a chest, you can specify how many items to withdraw by using a positive signed byte or how many to deposit using a negative signed byte. Finally, if you use the MMIO on a water tile, you will fully hydrate yourself.

You can also use `SUBMIT_BASE` on any tile within your base in order to mark the region as your base. If your base survives the night, then you will be awarded points as detailed in section 1.3, Items & Fort Grading.

After using any of these commands, you can read from `MMIO_STATUS` to see if they ran successfully. A value of 0 indicates success, and information on other possible values can be found in the starter MIPS code.

1.8 Day/Night

This game operates on a day/night cycle. During the day everything is peaceful, you can interact with blocks and build your fort without disrupting (ignoring the other player). When it turns to night, the screen will visibly darken; squirrels will spawn and the chase begins. The squirrels will walk towards the center and attempt to destroy anything getting in their way. Should your bot be unfortunate enough to come within 40 pixels of a squirrel, the squirrel will begin chasing after your bot and it will attack once within 16 pixels of your bot. After attacking, squirrels have a cooldown of 10,000 cycles giving your bot plenty of time to run away or seek shelter. When breaking a block, squirrels will deal 3 damage to the blocks durability every cycle. Once the durability reaches 0, the block will break.

Once the night is over, it will become day. The island flourishes again and every tile that is ground but was not originally, will return to its original state. Resources are again plentiful and you can repair any damage done to your fort and the cycle begins again.

If you would like to be notified when it becomes night time, you can either use the timer interrupt or the `REQUEST_NIGHT` interrupt.

1.9 Scoring

You can earn and lose points in a few ways.

- Dying without a bed costs 30 points
- Dying while still having a bed costs 10 points
- Destroying another player's bot earns 30 points
- Destroying a squirrel earns 5 points
- Breaking a block earns 1 point
- Placing a block earns 1 point
- Crafting an item earns some points (see Items & Fort Grading)
- Having items within a fort when night ends (see Items & Fort Grading for calculation)

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 generate **creativity**. The puzzle you have to solve is Count Disjoint Regions (FillFill for short). 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 FillFill

The puzzle is the same puzzle as from Lab7, 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 7.

Note that if you are making your own puzzle solver, the -debug will printout helpful info about why your solution was incorrect.

2.5.2 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:

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

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 generate creativity. 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 creativity**! To check if you were awarded creativity for your solution, read from **MMIO_STATUS** (`0xffff204c`). If reading from this address yields 0, your solution was correct. If it yields anything else, your solution was not correct.

Run with the `-debug` flag, request a puzzle, and submit something 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. For all intents and purposes, it is just a solution to Lab7 and can be greatly optimized to give your bot an edge over the competition.

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.

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:

Name	Register	Explanation
Status Register	$12	This register contains the interrupt mask and interrupt enable bits.
Cause Register	$13	This register contains the exception code field and pending interrupt bits.
Exception Program Counter (EPC)	$14	This register holds the PC of the executing instruction when the exception/interrupt occurred.

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.

Name	Interrupt Mask	Acknowledge Address
Timer	`0x8000`	`0xffff006c`
Bonk (wall collision)	`0x1000`	`0xffff0060`
Request Puzzle	`0x0800`	`0xffff00d8`
Request Night	`0x4000`	`0xffff00e0`

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

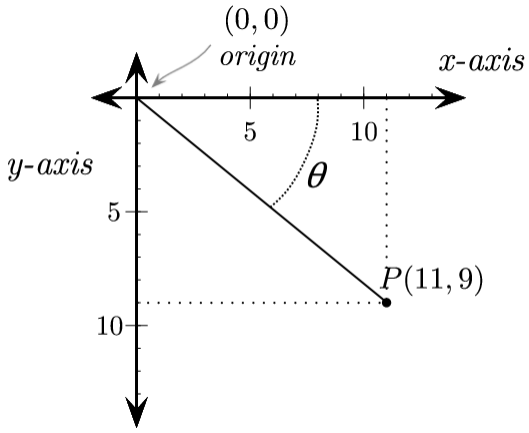
You will receive the **Request Puzzle** interrupt once the requested puzzle is ready to be written into the provided memory address. You must acknowledge this interrupt for the puzzle to be written to memory!

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° 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°.



P is at an angle of about 40°

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 (± 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

Argument	**Description**
`-file <file1.s> <file2.s> ...`	Specifies the assembly file(s) to use
`-file2 <file1.s> <file2.s> ...`	Specifies the assembly file(s) to use for a second SPIMBot
`-part1`	Run SPIMBot under Lab 9 part 1 conditions
`-part2`	Run SPIMBot under Lab 9 part 2 conditions
`-test`	Run SPIMBot starting with 65535 money. Useful for testing
`-debug`	Prints out scenario-specific information useful for debugging
`-limit`	Change the number of cycles the game runs for. Default is 10,000,000. Set to 0 for unlimited cycles
`-randommap`	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`
`-mapseed <seed>`	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`

`-randompuzzle`	Randomly generate puzzles with the current time as the seed. Note that this overrides `-puzzleseed`
`-puzzleseed <seed>`	Randomly generate puzzles based on the given seed. Seed should be a non-negative integer. Note that this overrides `-randompuzzle`
`-drawcycles <num>`	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
`-largemap`	Draws a larger map (but runs a little slower)
`-smallmap`	Draws a smaller map (but runs a little faster)
`-maponly`	Doesn't pop up the QtSpim window. Most useful when combined with `-run`
`-run`	Immediately begins the execution of SPIMBot's program
`-tournament`	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
`-prof_file <file>`	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
`-exit_when_done`	Automatically closes SPIMBot when contest is over
`-quiet`	Suppress extraneous error messages and warnings
`--version`	Prints the version of the binary (note the double-dash!) **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 70 points kernels in at least 3 games out of a total 4 games. **Note that the map seed controls the placement of squirrels, not the map itself!** This is the command we will use to run your code on some seed X for qualifications:

```
QtSpimbot -f spimbot.s -mapseed 233
```

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

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

Name	Address	Acceptable Values	Read	Write
VELOCITY	0xffff0010	-10 to 10	Current velocity	Updates velocity of the SPIMBot
ANGLE	0xffff0014	-360 to 360	Current orientation	Updates angle of SPIMBot when **ANGLE_ CONTROL** is written
ANGLE_ CONTROL	0xffff0018	0 (relative) 1 (absolute)	N/A	Updates angle to last value written to **ANGLE**
TIMER	0xffff001c	Anything	Number of elapsed cycles	Timer interrupt when elapsed cycles == write value
TIMER_ACK	0xffff006c	Anything	N/A	Acknowledge timer interrupt
BONK_ACK	0xffff0060	Anything	N/A	Acknowledge bonk interrupt
REQUEST_ PUZZLE_ACK	0xffff00d8	Anything	N/A	Acknowledge request puzzle interrupt
BOT_X	0xffff0020	N/A	Current X-coordinate, px	N/A
BOT_Y	0xffff0024	N/A	Current Y-coordinate, px	N/A
SCORES_ REQUEST	0xffff1018	Valid data address	N/A	M[address] = [your score, opponent score]
REQUEST_ PUZZLE	0xffff00d0	Valid data address	N/A	M[address] = new puzzle; sends Request Puzzle interrupt when ready

SUBMIT_ SOLUTION	`0xffff00d4`	Valid data address	N/A	Submits puzzle solution at M[address]
NIGHT_ACK	`0xffff00e0`	Anything	N/A	Acknowledge night interrupt
GET_MAP	`0xffff2040`	Valid data address	N/A	Writes `Map` struct to given memory location
GET_PUZZLE_ CNT	`N/A`	N/A	N/A	This command is not implemented.
GET_WOOD	`0xffff2000`	N/A	Number of wood in inventory	N/A
GET_STONE	`0xffff2004`	N/A	Number of stone in inventory	N/A
GET_WOOL	`0xffff2008`	N/A	Number of wool in inventory	N/A
GET_ WOODWALL	`0xffff200c`	N/A	Number of wood walls in inventory	N/A
GET_ STONEWALL	`0xffff2010`	N/A	Number of stone walls in inventory	N/A
GET_BED	`0xffff2014`	N/A	Number of beds in inventory	N/A
GET_CHEST	`0xffff2018`	N/A	Number of chests in inventory	N/A
GET_DOOR	`0xffff201c`	N/A	Number of doors in inventory	N/A

MMIO_STATUS	`0xffff204c`	N/A	Return status of previously used MMIO. See starter.s for more info.	N/A
BREAK_ BLOCK	`0xffff2020`	An integer of the format `0x0000XXYY` XX - column of block location YY - row of block location	N/A	Damages the block at tile [XX, YY] if it is breakable and the bot is in range (3 tiles euclidean distance)
CRAFT	`0xffff2024`	Any valid item id (0 to 8)	N/A	Crafts the item id specified. If you have insufficient resources for this, it will fail and output a debug message
ATTACK	`0xffff2028`	An integer of the format `0x0000XXYY` XX - column of block location YY - row of block location	N/A	Kills all squirrels in the specified tile (if in 3 tiles euclidean distance) and deals 40 damage to the opponent if they are in that tile
PLACE_BLOCK	`0xffff202c`	A number of the form `ttttXXYY`. tttt - item id XX - column of block location YY - row of block location	N/A	Places a block specified by the item id on tile [XX, YY]. Fails if the tile is not within 3 tiles euclidean distance, if the player doesn't have the tile, or if the target position does not contain Floor/Ground.

USE_BLOCK	`0xffff2030`	A number of the form `0xnnttXXYY`. nn - signed quantity of items tt - item id XX - column of block location YY - row of block location If the tile selected is not a chest, then the top 16 bits are ignored.	N/A	Uses the block. The behavior of this differs based on the block being operated on. Fails if you are not in range (3 tiles euclidean distance). Exact details about behavior per block is defined in The Spimbot section.
GET_ INVENTORY	`0xffff2034`	Valid data address	N/A	M[address] = [unsigned; 8] 8 item ids are written to address to address + 32 in the order: Wood, Stone, Wool, Wood Wall, StoneWall, Bed, Chest, Door
GET_ SQUIRRELS	`0xffff2038`	Valid data address	N/A	M[address] = struct SquirrelList See Appendix for struct definition
SUBMIT_BASE	`0xffff203c`	An integer of the format `0x0000XXYY` XX - column of block location YY - row of block location	N/A	Base at the given location is submitted and graded

GET_ HYDRATION	`0xffff2044`	N/A	Gets your spimbot's hydration level. 100 is fully hydrated and 0 is dehydrated.	N/A
GET_HEALTH	`0xffff2048`	N/A	Gets your spimbot's health. 100 is completely healthy and 0 is dead.	N/A

Appendix B: Struct Definitions

```
struct Map {
  int map[40][40]; // map[a][b] = 0xDDDDtobw
  // DDDD = (int)(durability * 0.1)
  // t - block type
  // o - owner. 0 if player 0 (red). 1 if player 1 (blue). -1 if nobody.
  // b - breakable
  // w - walkable
}

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

struct SquirrelList {
    int count;
    struct Squirrel mobs[100];
}

struct Squirrel {
    int x_tile;
    int y_tile;
    int health;
};
```

Appendix C: 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 - computes sqrt(x^2 + y^2)
# $a0 - x
# $a1 - y
# returns the distance
# ------------------------------------------------------------------------
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
```