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

Contents

0 Document Updates

1 The Game

1.1 The Map

1.2 Tile Types

1.3 Items

1.4 Recipes

1.5 The Objective

2 SPIMBot I/O

2.1 Orientation Control

2.2 Odometry

2.3 Bonk (Wall Collision) Sensor

2.4 Timer

2.5 Puzzle

2.5.1 FillFill

2.5.2 Puzzle Struct

2.5.3 Requesting a Puzzle

2.5.4 Submitting Your Solution

2.5.5 The Slow Solver

2.5.6 Debugging

3 Interrupts
0 Document Updates

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

1. Nothing yet!
1 The Game

For this year’s SPIMBot competition, you will be writing code to control a bot to survive 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]).

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:
<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>Forest</td>
<td>🌳</td>
<td>0</td>
<td>Yes</td>
<td>Breaking this block yields wood.</td>
</tr>
<tr>
<td>Sheep</td>
<td>🐐</td>
<td>2</td>
<td>Yes</td>
<td>Breaking this block/sheep yields wool.</td>
</tr>
<tr>
<td>Stone</td>
<td>🍀</td>
<td>4</td>
<td>Yes</td>
<td>Breaking this block yields wool.</td>
</tr>
<tr>
<td>Floor/Ground</td>
<td>🕊</td>
<td>8</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Water</td>
<td>🌊</td>
<td>9</td>
<td>Yes</td>
<td>Using this block allows you to drink</td>
</tr>
</tbody>
</table>

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 Items

Items in this game are either dropped from blocks when broken or they are crafted using other items.

<table>
<thead>
<tr>
<th>Tile type</th>
<th>Tile Code</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wood</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Stone</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Wool</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

### 1.4 Recipes

There is currently only one recipe in the game; but once lab Spimbot is released, all the recipes for crafting things like beds and doors will be released. In Lab 9, you can only craft a single item
that requires 1 wool, 1 stone, and 1 wood.

1.5 The Objective

Your goal is simple: gather resources, craft them, and survive! Scattered across tiles that when broken or used, yield resources that can be used in order to protect yourself. For Lab 9, you will be playing a stripped down single-person version of the full game. For Lab 9, the goal is to simply collect the necessary resources and then craft. In the full-game, you will need to survive the night and build forts in order to protect yourself from mobs.

Another important aspect of the game is solving puzzles. Solving a puzzle gets you 1 creativity. Creativity doesn’t contribute to your score (except in lab 10). Creativity is needed (along with resources) to craft items. More info about the puzzles can be found in Section 2.5.
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:

```c
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!

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. 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.

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 showing angle and position](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 \texttt{ANGLE} and \texttt{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 \texttt{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>-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 9 part 1 conditions</td>
</tr>
<tr>
<td>-part2</td>
<td>Run SPIMBot under Lab 9 part 2 conditions</td>
</tr>
<tr>
<td>-test</td>
<td>Run SPIMBot starting with 65535 money. Useful for testing</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;seed&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>Option</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</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;seed&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 &lt;num&gt;</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>-smallmap</td>
<td>Draws a smaller map (but runs a little faster)</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. Also forces the map and puzzle seeds to be random. This includes disabling error, which can make debugging more difficult.</td>
</tr>
<tr>
<td>-prof_file &lt;file&gt;</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>
<tr>
<td>--version</td>
<td>Prints the version of the binary (note the double-dash!). Latest version: 2020–12–2</td>
</tr>
</tbody>
</table>

**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 ??.

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 ANGLE_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 angle to last value written to ANGLE</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>BOT_X</td>
<td>0xffff0020</td>
<td>N/A</td>
<td>Current X-coordinate, px</td>
<td>N/A</td>
</tr>
<tr>
<td>BOT_Y</td>
<td>0xffff0024</td>
<td>N/A</td>
<td>Current Y-coordinate, px</td>
<td>N/A</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>Function</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>SUBMIT_SOLUTION</td>
<td>0xffff00d4</td>
<td>Valid data address</td>
<td>Submits puzzle solution at M[address]</td>
<td></td>
</tr>
<tr>
<td>GET_MAP</td>
<td>0xffff2040</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 WOOD</td>
<td>0xffff2000</td>
<td>N/A</td>
<td>Number of wood in inventory</td>
<td></td>
</tr>
<tr>
<td>GET STONE</td>
<td>0xffff2004</td>
<td>N/A</td>
<td>Number of stone in inventory</td>
<td></td>
</tr>
<tr>
<td>GET WOOL</td>
<td>0xffff2008</td>
<td>N/A</td>
<td>Number of wool in inventory</td>
<td></td>
</tr>
<tr>
<td>BREAK BLOCK</td>
<td>0xffff2020</td>
<td>An integer of the format 0x0000XXYY</td>
<td>Breaks the block (if able to) at the row and column if you are in range (3 tiles euclidean distance)</td>
<td></td>
</tr>
<tr>
<td>CRAFT</td>
<td>0xffff2024</td>
<td>Any valid item id (0 to 8)</td>
<td>Crafts the item id specified. If you have insufficient resources for this, it will fail and output a debug message</td>
<td></td>
</tr>
<tr>
<td>Command</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>USE_BLOCK</td>
<td>0xffff2030</td>
<td>A number of the form 0xnnnttXXYY.                                                                ían - 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, 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).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GET_INVENTORY</td>
<td>0xffff2034</td>
<td>Valid data address</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>SUBMIT_BASE</td>
<td>0xffff203c</td>
<td>An integer of the format 0x0000XXYY .toast - column of block location YY - row of block location</td>
<td>N/A</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] = tile_id
}

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

// 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 - 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