"Pew Pew Pew!" – A SPIMBot

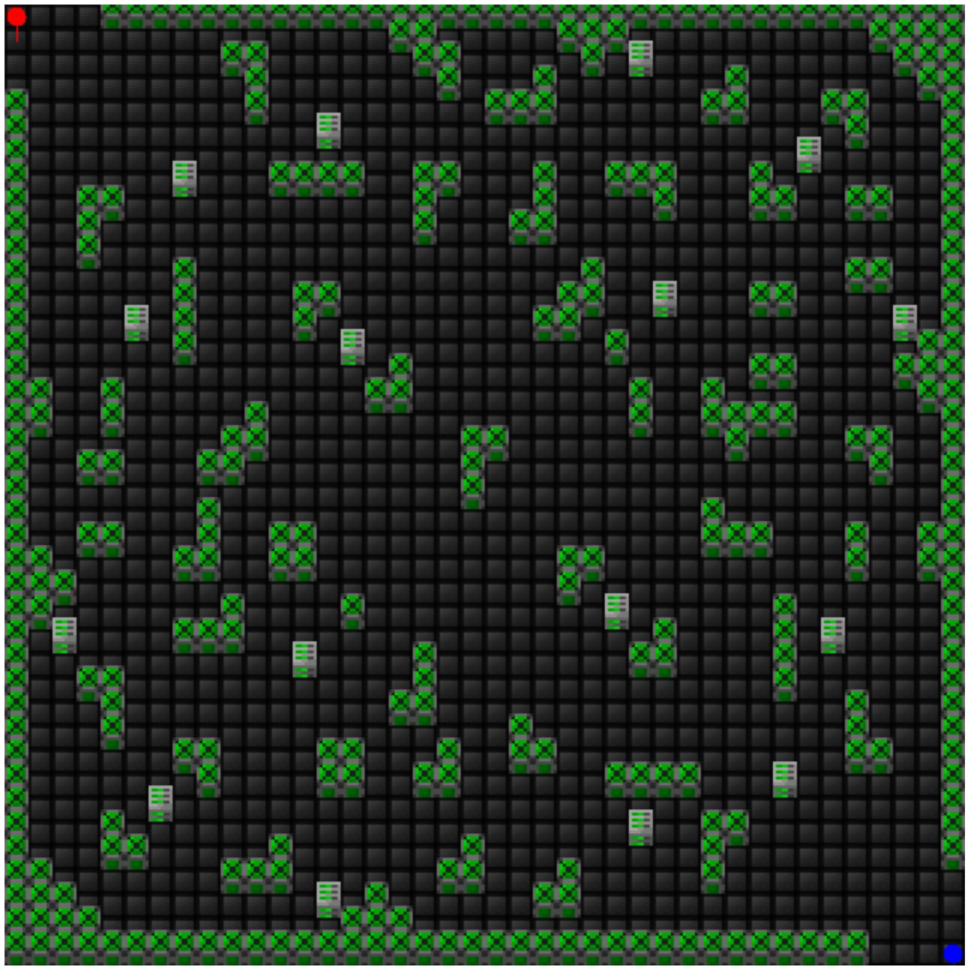
Contents

1 The Game

For this year's SPIMBot competition, you will be writing code to control a bot to shoot and capture hosts on the map, as well as shoot your opponent for extra points. The hosts will spawn in fixed locations, but there will be random obstacles scattered around the map, and you won't be able to see where your opponent is moving! Good luck!

1.1 The Map

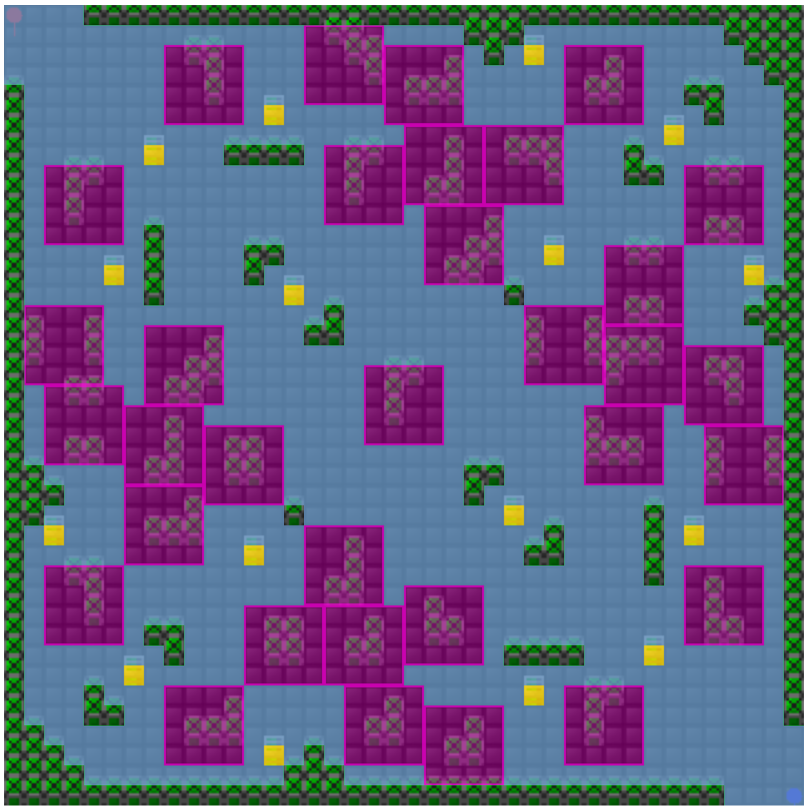
The SPIMBot map looks like this:



Bot 1 (red) always starts in the top left corner (Tile [0, 0], pixels [4, 4]), and bot 2 (blue) always starts in the bottom right corner (Tile [31, 31], pixels [316, 316]). Randomly generated obstacles appear on the map, and hosts will spawn at predetermined locations. Each tile is 8 pixels by 8 pixels wide.

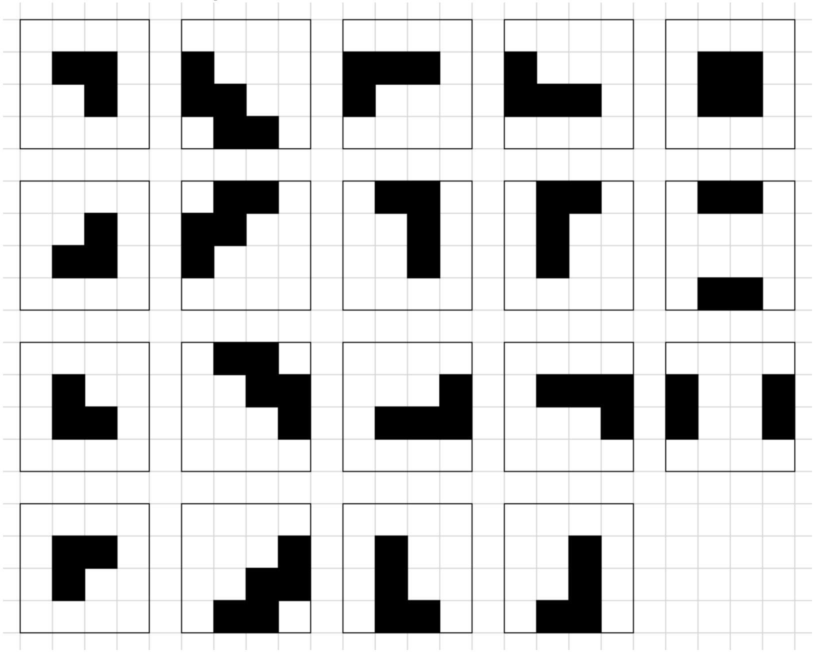
Note: During grading and tournament qualification, your SPIMBot will always be bot #1.

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



Visually, the space guaranteed to be available for movement (non-obstacles) is highlighted in light blue. The light pink squares are spaces where obstacles will spawn, and the light yellow squares are the fixed host positions. Although the hosts and walls are drawn 1.5 tiles tall, they have a 1 square tile hitbox.

The various 4x4 squares around the map may contain obstacles, and are randomly chosen with equal weighting from the following pool:



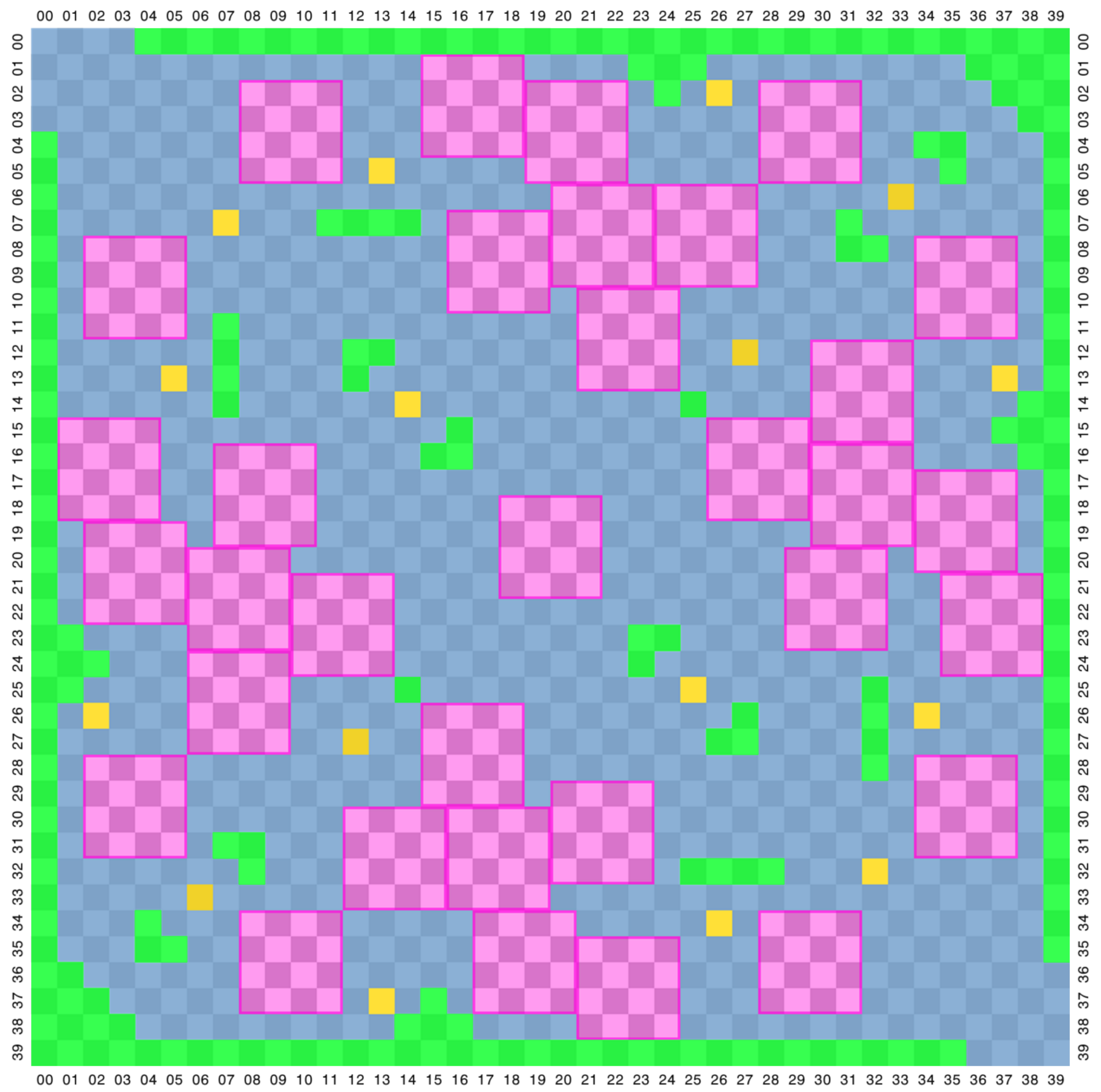
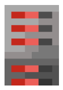
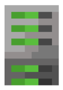
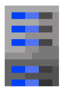


Figure 1: The same information as the image on the previous page, except on a nice, white and gray background with numbers labelling the rows and columns!

1.2 Tile types

Below is a summary of the different tile types and their properties.

Tile type	Image	Can Collide with Player	Can Collide with UDP Packet / Scanner Beam
Floor		No	No
Wall		Yes	Yes
Red Host		No	Yes
Neutral Host		No	Yes
Blue Host		No	Yes

For more details on how to interact with and capture hosts, see section SPIMBot Physics.

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

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, or
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 SPIMBot clockwise. While it may not sound intuitive, this matches the normal Cartesian coordinates (in that the +x direction is angle 0, +y=90, -x=180, and -y=270), since we consider the top-left corner to be (0,0) with +x and +y being right and down, respectively. 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 Map

Writing a pointer to **ARENA_MAP** will cause the map layout to be written to the pointer. The layout will look like this:

```
#define NUM_ROWS 40
#define NUM_COLS 40

struct arena_map {
    char map[NUM_ROWS][NUM_COLS];
};
```

Each byte represents a single square tile on the board, following these rules:

```
#define WALL            1
#define HOST_MASK       2
#define FRIENDLY_MASK   4
#define ENEMY_MASK      8
```

where the resulting tile type is formed by the bitwise OR of the bitmasks.

For example, a ground tile shows up as a 0, a wall tile shows up as a 1, a neutral host shows up as a 2, and an enemy host is 10 (8 | 2). **Your opponent will not show up on the map.**

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

2.6 Opponent Location Hint

Writing a pointer to **GET_OPPONENT_HINT** (0xffff00ec) will cause a hint about your opponent's position to be written to the pointer. The layout will look like this:

```
struct OpponentHintInfo {
    // X tile of the host you own that is closest to your opponent
    char host_x;

    // Y tile of the host you own that is closest to your opponent
    char host_y;

    // Angle from the target host to your opponent, -360 to +360
    short angle;
};
```

If you don't own any hosts, all three fields will be -1.

2.7 UDP Packet (Laser)

Writing anything (including $0) to **SHOOT_UDP_PACKET** (0xffff00e0) will attempt to shoot a UDP Packet from your SPIMBot. The shot starts at your bot's location and travels in the direction that the bot is currently travelling and costs 50 bytecoins. If your bot has insufficient bytecoins, nothing happens.

More details about the UDP Packet can be found in the **Spimbot Physics** section.

2.8 Scanner

Writing a pointer to **USE_SCANNER** (0xffff00e8) will fire your SPIMBot's scanner. Doing so costs 1 bytecoin.

The scanner traces a line 20 tiles long in the direction your robot is facing, and returns information about the first thing that it hits. The scanner beam does not go through walls. The layout will look like this:

```
#define WALL            1
#define HOST_MASK       2
#define FRIENDLY_MASK   4
#define ENEMY_MASK      8
#define PLAYER_MASK     16

struct ScannerInfo {
    // X tile of the tile that was "hit".
    unsigned char hit_x;

    // Y tile of the tile that was "hit".
    unsigned char hit_y;

    char tile_type;
};
```

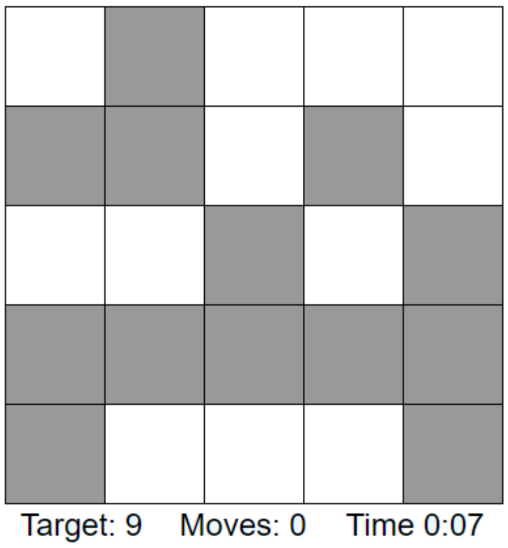
The scanner can hit walls, hosts (friendly or not), and bots. The returned (x, y) coordinate is in tile coordinates, not pixel coordinates! The tile type field is filled out like the similar field in the **ARENA_MAP** call, except there is the additional possibility of hitting a player.

If the scanner beam expires (travels 20 tiles without hitting a wall, beacon, or your opponent), the last tile that the beam travelled to is returned and the tile type is set to GROUND (0).

2.9 Puzzle

Mining (solving puzzles) is the only way to earn bytecoins. In this SpimBot Lab, the puzzle is LightsOut. It is similar to the one in Lab 7 except we are not giving clues this time. **Your Lab 7 solver will not work without modification! Use the provided solution instead!**

1. **The LightsOut puzzle**
   LightsOut is a puzzle involving a grid of lights. LightsOut is based on a deceptively simple concept. Clicking on a cell toggles that cell and each of its orthogonal neighbours. The goal is to turn out all the lights. You can find an example of the puzzle here: `https://www.logicgamesonline.com/lightsout/`. Below is an example of a puzzle board.



   For our SpimBot, lights have more colors! All lights follow the same sequence of colors. Toggling the lights will advance them into their next color. The sequence always starts with the cool "black color". When the last color is reached, toggling it turns it back to black.

2. **Puzzle Struct**
   The puzzles come in a struct. The struct written to memory is defined as follows:

```
#define MAX_GRIDSIZE 16
typedef struct LightsOut {
    int num_rows;
    int num_cols;
    int num_colors;
    unsigned char board[MAX_GRIDSIZE*MAX_GRIDSIZE];
} LightsOut;
```

   Note that we are not giving clues, as opposed to Lab7.
   A list of possible puzzle dimensions can be found in Appendix D.

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 the puzzle will not be generated immediately. Instead, 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 and received them.

4. **Submitting Your Solution**
   After solving the puzzle, you need to submit the solution to earn bytecoins. To submit your solution, you will simply write a pointer of the your solution board to **SUBMIT_SOLUTION**. If your solution is correct, you will be rewarded with bytecoins.

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 recursive backtracking algorithm. It tries all possible combinations of actions in the first row, use "chase the light" strategy for the remaining rows, and return the solution once all lights are off.
   ```
   /* puzzle - pointer to puzzle
      solution - solution will be stored here after function call
      row - row number of current cell, start with 0
      col - column number of current cell, start with 0 */
   solve(LightsOut* puzzle, unsigned char* solution, int row, int col)
   ```
   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.

6. **Rewards**
   You will receive 50 bytecoins upon successfully solving a puzzle.

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
Respawn	0x2000	0xffff00f0

3.2 Bonk

You will receive the **Bonk** interrupt if your robot runs into a wall. Your robot'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!

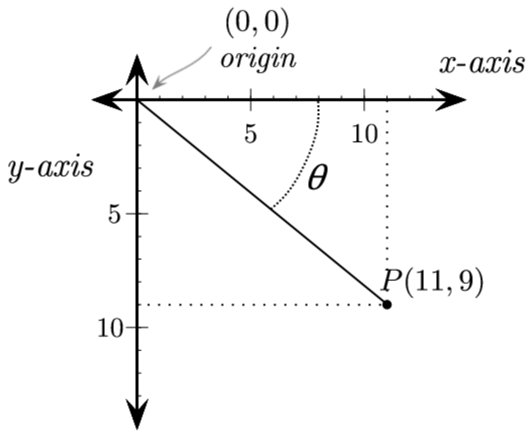
3.4 Respawn

You will receive the **Respawn** interrupt if your bot is hit by your opponent's UDP packet and respawns in a different location, with its velocity set to zero.

4 SPIMBot Physics

4.1 Position and Velocity

In the SPIM Universe, positions are given in pixels, or px. Pixels start in the upper left at (X=0, Y=0) and end in the bottom right at (X=320, Y=320). Angles start at 0° (pointing down the positive x-axis, or right) and sweep clockwise (90° points down).



The center of the SPIMBot dot is where you are positioned. The SPIMBot itself is a circle around its position, 6 pixels in diameter.

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 acceleration, angular or linear. This means that you can rotate the SPIMBot instantly by writing to the **ANGLE** and **ANGLE_CONTROL** MMIO addresses (See the section in **SPIMBot IO** for more details). This is useful for aiming UDP packets and scanner beams!

4.2 Collisions

If your position is about to go out-of-bounds (beyond 0 or 320 on any 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; i.e. SPIMBots are points with respect to wall collisions.

Note (2): Hosts are opaque to UDP Packets and scanners, but transparent to SPIMBots! You will not bonk or stop when you enter a cell occupied by a host.

With respect to UDP Packets and scanner beams, the SPIMBot is a circle with diameter 6px centered at its position. If a scanner beam hits any part of this circle, it will report that it hit a SPIMBot; if an enemy UDP packet hits any part of this circle, the hit SPIMBot will crash and be forced to respawn (See section **Respawn mechanics** for more details).

4.3 UDP Packets

UDP Packets are what your SPIMBot uses to interact with the map and with the opponent. They can be fired by writing to the **SHOOT_UDP_PACKET** MMIO address (See the section in **SPIMBot IO** for more details).

UDP Packets travel at 150 velocity units, 15 times faster than the maximum speed of the SPIMBot. They have zero width, and can travel a maximum of 15 tiles.

UDP Packets can collide with walls, hosts, or SPIMBots other than the SPIMBot that fired it. If a UDP Packet hits an enemy host, it becomes a neutral host; If it hits a neutral host, it becomes a friendly host; If it hits the opponent, the opponent crashes and has to reboot. (See the next section on respawn mechanics.)

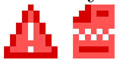
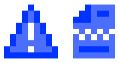
4.4 Respawn mechanics

When your SPIMBot is hit by an enemy UDP Packet, it crashes and has to reboot. Rebooting follows this algorithm:

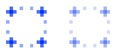
1. If you own any hosts, respawn at the host you own that is the closest to but not less than 15 tiles away from your opponent.
2. If you do not own any hosts, and there are neutral hosts on the map, respawn at the neutral host that is closest to but not less than 15 tiles away from your opponent. The neutral host becomes a friendly host.
3. Otherwise, convert the host that is closest to but not less than 15 tiles away from your opponent into a friendly host, and respawn at it.
4. Finally, set the SPIMBot velocity to zero.

Be sure to enable the Respawn interrupt to be notified of when you have respawned.

When you crashes and reboot, these two images will flash at the location you crashed.

 or 

These two images will flash at the location you rebooted.

or 

5 Scoring

For the tournament, your bot will be awarded points during the match two different ways:

1. Fifteen (15) points each time you successfully hit your opponent with a UDP packet;
2. One (1) point for each host you own, checked every 250,000 clock cycles. This will trigger 40 times per game.
   Note: This is checked every 250,000 clock cycles, globally, not every 250,000 cycles you own the host. So all hosts update simultaneously and award points to whichever SPIMBot controls it at that instant in time.

Whoever has more points at the end of a round wins the round. In the case of a tie, a winner is chosen randomly.

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

6 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.

6.1 Useful Command Line Arguments

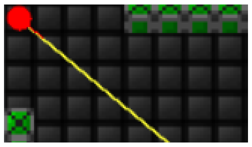
Argument	**Description**
`-file <file1.s> <file2.s> ...`	Specifies the assembly file(s) to use
`-file2 <file1.s> <file2.s> ...`	Specifies the assembly file(s) to use for a second SPIMBot
`-part1`	Run SPIMBot under Lab 9 part 1 conditions
`-part2`	Run SPIMBot under Lab 9 part 2 conditions
`-test`	Run 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

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!

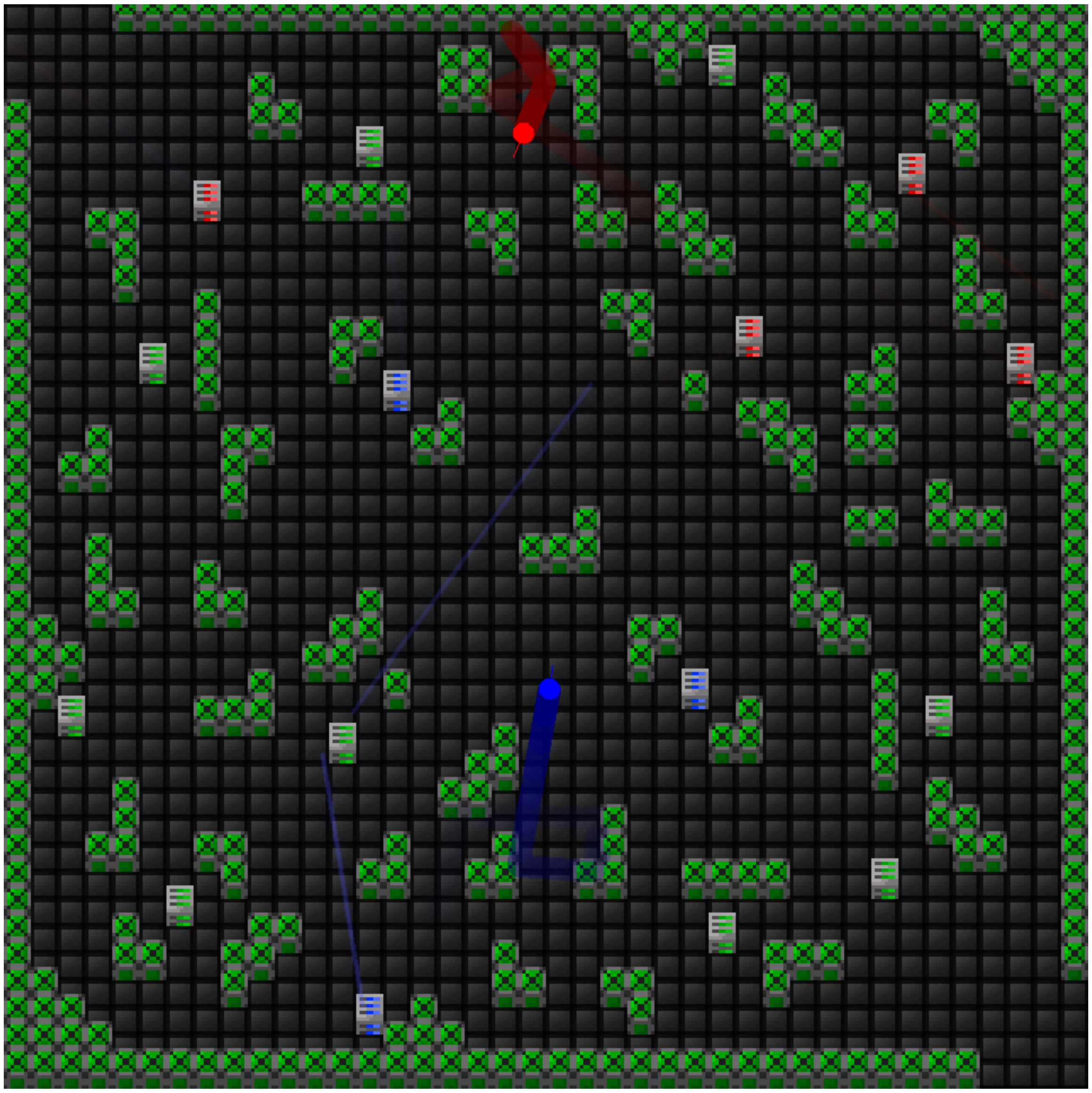
Note that `-randommap` and `-mapseed` override one another, and that `-randompuzzle` and `-puzzleseed` override one another.The `-tournament` flag also overrides most other flags. The flag that is typed last will be the overriding flag.

Run with `-debug` to see traces from the scanner!



6.2 Fun Command Line Arguments

Try running with the `-largemap` flag!



7 Tournament Rules

7.1 Qualifying Round

To qualify for the SPIMBot tournament, you must be able to outperform the baseline bot we have provided in `adversary.s`. You must be able to beat the baseline bot by at least a 200 point margin in at least 2 games out of a total 3 games. Each of these 3 games will use a different puzzle and map seed. This is the command we will use to run your code on some seed X for qualifications: `QtSpimbot -file spimbot.s -file2 adversary.s -puzzleseed X -mapseed X -pt3`

For more details on how your bot will be scored in these games, see **Section 5: Scoring**.

7.2 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!

7.3 LabSpimbot Grading

LabSpimbot grade breakdown is specified in the LabSpimbot handout.

8 Appendix A: MMIO

Name	**Address**	**Acceptable Values**	**Read**	**Write**
VELOCITY	0xffff0010	-10 to 10	Current velocity	Updates velocity
ANGLE	0xffff0014	-360 to 360	Current orientation	Updates angle 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
RESPAWN_ACK	0xffff00f0	Anything	N/A	Acknowledge respawn 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]
ARENA_MAP	0xffff00dc	Valid data address	N/A	M[address] = tile map

SHOOT_UDP_PACKET	0xffff00e0	Anything	N/A	Shoot a UDP Packet, costs 50 bytecoins
GET_BYTECOINS	0xffff00e4	N/A	Returns how many bytecoins you have	N/A
USE_SCANNER	0xffff00e8	Valid data address	N/A	Shoot a scan beam, costs 1 bytecoins. M[address] = result
GET_OPPONENT_HINT	0xffff00ec	Valid data address	N/A	M[address] = hint to opponent's location

9 Appendix B: Type Definitions

```
#define NUM_ROWS 40
#define NUM_COLS 40

#define WALL            1
#define HOST_MASK       2
#define FRIENDLY_MASK   4
#define ENEMY_MASK      8

/**
 *  Array with one entry for each tile on the board.
 *  See section SPIMBot IO for more details.
 */
struct ArenaMap {
    char map[NUM_ROWS][NUM_COLS];
};

/**
 *  Struct for receiving hints about your opponent's position.
 *  See section SPIMBot IO for more details.
 */
struct OpponentHintInfo {
    // X tile of the host you own that is closest to your opponent
    char host_x;

    // Y tile of the host you own that is closest to your opponent
    char host_y;

    // Angle from the target host to your opponent, -360 to +360
    short angle;
};

/**
 *  Struct for receiving data from your SPIMBot's scanner.
 *  See section SPIMBot IO for more details.
 */
#define PLAYER_MASK     16
struct ScannerInfo {
    // X tile of the tile that was "hit".
    unsigned char hit_x;

    // Y tile of the tile that was "hit".
    unsigned char hit_y;

    char tile_type;
};
```

10 Appendix C: Cost

Action	Bytecoins
Solve puzzle	+50
Use Scanner	-1
Shoot UDP Packet	-50

11 Appendix D: Puzzle Dimensions

The dimensions of each LightsOut puzzle are selected from the following list uniformly with a weight.

num_rows	num_cols	num_colors	weight
12	4	2	2
15	4	2	2
16	4	2	2
6	5	2	1
13	5	2	2
5	6	2	1
7	3	3	1
12	3	3	2
13	3	3	2
11	5	3	1