SPIMbot documentation

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

SPIMbot Input/Output

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

Name	Address	Acceptable Values/Range	Read	Write
VELOCITY	`0xffff0010`	10 to -10	Gives your SPIMbot's current velocity.	Immediately updates your SPIMbot's velocity.
ANGLE	`0xffff0014`	-360 to 360	Gives your SPIMbot's current orientation.	No immediate effect; the value written is used when **ANGLE_CONTROL** is written.
ANGLE_CONTROL	`0xffff0018`	0 *(relative)*, 1 *(absolute)*	N/A	Updates your SPIMbot's orientation, using the last value written to **ANGLE** as a relative or absolute angle (depending on the value written to this address).
TIMER	`0xffff001c`	0 to `0xffffffff`	Gives the number of elapsed cycles.	Requests a timer interrupt at the cycle written.
BOT_X	`0xffff0020`	0 to 300	Gives your SPIMbot's current X location.	N/A
BOT_Y	`0xffff0024`	0 to 300	Gives your SPIMbot's current Y location.	N/A
OTHER_BOT_X	`0xffff00a0`	N/A	Gives your opponent SPIMbot's X location.	N/A
OTHER_BOT_Y	`0xffff00a4`	N/A	Gives your opponent SPIMbot's Y location.	N/A

FRUIT_SCAN	`0xffff005c`	*any valid data address*	N/A	Scans for all fruits eligible to be caught and writes the data to the address provided in a 2D array of fruit_information_t structs. The array will be NULL terminated and will be no larger than 260 bytes.
FRUIT_SMASH	`0xffff0068`	*any*	N/A	Attempts to smash the most recently smooshed fruit. Only works when at the bottom edge of the map and when inside the bonk interrupt handler (**b efore** acknowledging the bonk interrupt).
GET_ENERGY	`0xffff00c8`	0 to 1000	Current energy level of the bot.	N/A
REQUEST_PUZZLE	`0xffff00d0`	any valid data address	N/A	Returns the puzzle grid. Format: # rows, followed by, # columns, followed by puzzle grid, stored one row at a time. The grid rows and columns will each range between 16 and 64, inclusive.
SUBMIT_SOLUTION	`0xffff00d4`	any valid data address	N/A	Submits a pointer to the first node in your solution linked list.
REQUEST_WORD	`0xffff00dc`	any valid data address	N/A	Returns the word that should be looked for in the puzzle grid. The word is NULL terminated and is no larger than 80 characters.
~~**SCORES_REQUEST**~~	`0xffff1018`	*any valid data address*	N/A	Writes the score information to the provided address. The score information will be an array of 2 integers where your score is the first element.
PRINT_INT	`0xffff0080`	*any*	N/A	Prints an integer to the screen. (useful for debugging).
PRINT_FLOAT	`0xffff0084`	*any*	N/A	Prints a floating point value to the screen. (useful for debugging).
PRINT_HEX	`0xffff0088`	*any*	N/A	Prints an integer to the screen in hexadecimal notation. (useful for debugging).

We describe these in more detail below.

Orientation Control

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

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

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

Bonk (Wall Collision) Sensor

The bonk sensor signals an interrupt whenever SPIMbot runs into a wall. Note that SPIMbot's velocity is set to zero when it hits a wall. (See below how to register for and acknowledge interrupts.)

Timer

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

Fruit Scanning

You'll need to make space for the array of fruit information. When you write the address of this space to **FRUIT_SCAN**, that space will be populated with a `NULL` terminated array of `fruit_information_t` structs representing all the fruits on the screen at that time (the word immediately after the last `fruit_information_t` struct ends will be `NULL`). Another way of thinking about this is that the last fruit's id will be 0.

Fruits Smashing

Whenever you write to this address, SPIMbot will attempt to smash the most recently smooshed fruit. This will only work if the following conditions are met:

- There is at least 1 smooshed fruit to smash
- SPIMbot is currently handling the bonk interrupt
- SPIMbot is currently at the bottom edge of the map

Note that you are considered to be in the bonk interrupt handler if the bonk interrupt bit in the Cause register is 1. Since acknowledging the bonk interrupt will set this to 0, **you should not acknowledge the bonk interrupt until you are done smashing fruits.**

Puzzle

You will need to make space for the puzzle grid and word. When you write the address of the grid space into **REQUEST_PUZZLE** the process of having the puzzle written into that memory space will be initiated and an interrupt will fire when the puzzle is ready. The grid is organized as follows:

- Number of rows (anywhere between 16 and 64 inclusive)
- Number of columns (anywhere between 16 and 64 inclusive)
- Grid of characters, stored one row at a time

Then, you can write the address of the word space into **REQUEST_WORD**, and the word you need to find will be written into that address space. The maximum number of characters in that word is 80 characters, and it is NULL terminated.

To solve the puzzle, you will need to allocate a node_memory address space and a new_node_address space, as demonstrated in lab 8. The node_memory space will be used to store the linked list of your solution. Each node in the linked list should contain the location (row and column) of the respective letter in the grid followed by a pointer to the next letter of the word.

Finally, to submit the puzzle solution, you should submit the address of the first node in the linked list into the **SUBMIT_SOLUTION** memory mapped address.

OTHER_BOT_X, OTHER_BOT_Y

Reading from these addresses gives you the location of the enemy bot. However, note that these values only get "refreshed" once every 5000 cycles in order to prevent one bot from exactly following the other bot.

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.

Interrupt acknowledgment

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

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

Name	Interrupt Mask	Acknowledge Address
Timer	`0x8000`	`0xffff006c`
Bonk (wall collision)	`0x1000`	`0xffff0060`
Fruit smooshed (fruit collision)	`0x2000`	`0xffff0064`
Out of energy	`0x4000`	`0xffff00c4`
Request puzzle	`0x800`	`0xffff00d8`

Fruit smooshed

This interrupt happens whenever a fruit gets smooshed to your bot (ie collides with and gets stuck to your bot). Note that even if you don't choose to accept this interrupt, the fruit will be smooshed anyways. We guarantee that the number of fruit smooshed interrupts you receive will be exactly the same as the number of fruits that get smooshed against your bot.

Request puzzle

This interrupt happens whenever the requested puzzle's grid has finished getting copied into the bot's memory and is now ready to be read.

Out of energy interrupt

Although you can check your energy level at any time by loading from **GET_ENERGY**, you can also rely on the **OUT_OF_ENERGY** interrupt to signal your bot that it is out of energy. This interrupt is delivered when your energy level hits zero. If your energy level remains zero after receiving the interrupt, you will not receive the interrupt again. Only after gaining some energy will it be possible to receive the interrupt again.

Running and Testing Your Code

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

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

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

During the tournament, we'll run with the following parameters: `-maponly -run -tournament -randommap -largemap -exit_when_done`

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
`-maponly`	Doesn't pop up the QtSpim window
`-run`	Immediately begins the execution of SPIMbot's program
`-largemap`	Draws a larger map (but runs a little slower)
`-debug`	prints out scenario-specific information useful for debugging
`-prof_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
`-tournament`	A command that disables the console, SPIM syscalls, and some other features of SPIM for the purpose of running a smooth tournament
`-randommap`	If the scenario supports it, giving this flag indicates that a random number generator should be used to assign starting state for the scenario
`-randomseed` ***`num`***	This option allows you to control the seed given to the random number generator
`-drawcycles` ***`num`***	Causes the map to be redrawn every *num* cycles. The default is `0x2000`, and lower values slow execution down, allowing movement to be observed much better
`-exit_when_done`	Automatically closes SPIMbot when contest is over
`-quiet`	Suppress extraneous error messages and warnings

Source Code:

spimbot.fa15.nov23.tar.gz