Build a Simple-Simon Game using Flea-Scope™

# Introduction

Hopefully you remember the game -- four buttons, four different color lights, and four different sounds, and you have to repeat the “challenge” pattern as it gets longer and longer, and if you fail, you get to hear the “raspberry” sound!

Using a Flea-Scope™ board running StickOS® BASIC, it is possible to quickly build a simple-simon game in minutes at home, *using nothing but a soldering iron and a web browser! (Or you can even build it without a soldering iron, using a solderless breadboard!)*

|  |  |
| --- | --- |
|  |  |

***Figure 1a and 1b -- Flea-Scope™ wired for Simple-Simon***

A Flea-Scope can be controlled by *any Chromium-based web browser* that supports WebUSB API or Web Serial API running on a computer, tablet, or phone (!!!), with no need for further software install – ***just plug it in and open a web-page and you are ready to write and debug your simple-simon game in StickOS BASIC.***

StickOS BASIC running within the Flea-Scope is an *entirely MCU-resident interactive programming environment*, which includes an easy-to-use editor, transparent line-by-line compiler, interactive debugger, performance profiler, and flash filesystem, all controlled thru an interactive command-line user interface. In StickOS, external MCU pins may be mapped to BASIC “pin variables” for manipulation or examination, and internal MCU peripherals may be managed by BASIC control statements and interrupt handlers.

You can find a short video of the simple-simon game in action here:

<https://1drv.ms/v/s!An6qoNgNXmeQhbQsx2NcGufonJJkBA?e=EPw5kv>

You can find an introduction to StickOS BASIC here, in *Microcontrollers For Everyone!*:

<https://rtestardi.github.io/pages/mfe.pdf>

# Simple-Simon Game State Diagram

This game is actually quite a clever piece of programming, even though the original game was undoubtedly written for a 4-bit microprocessor! It involves a complex state machine shown in Figure 2 whose decomposition takes a bit of time to understand...

In the state diagram, there are only three main game “states”, represented by ovals:

* **challenge** (the game is playing the notes for the user)
* **confirm** (the user has to repeat the notes back to the game -- if successful, the challenge grows by one note)
* **raspberry** (the user failed -- the game is over and will start from the beginning again)
* (in the code, a final state is used just to restart the game after a delay, so the game plays over and over)

There are four “events” that can occur while in a game state, possibly affecting “state variables” or leading you to a new game “state”:

* **start** (occurs at the start of the game and leads us to the “challenge” state)
* **timer** (occurs every 500ms)
* **button down** (occurs when the user presses a button)
* **button up** (occurs when the user releases a button)

When an event occurs, you follow the flowchart decision tree (rectangles for actions and diamonds for decisions) using or modifying the other “state variables” and eventually end up at a new game “state”.

The “state variables” which contain game state beyond the three main game states include:

* arrayLength -- number of notes in the noteArray
* noteArray[] -- 1..4 in each slot, for colors 1..4
* arrayIndex -- 0..arrayLength-1 (used for challenge and confirm)
* gameMode -- 0 = challenge; 1 = confirm; 2 = raspberry
* replayState -- 0 = note; 1 = quiet
* waitTime -- 0..4 (timer ticks)

**timer**

audio play noteArray[arrayIndex];  
replayState = 1;

replayState

0

1

audio play off;  
replayState = 0;  
increment arrayIndex;

arrayIndex = arrayLength

yes

gameMode = 1;  
arrayIndex = 0;  
waitTime = 0;

no

noteArray[arrayLength] = random;  
increment arrayLength;  
arrayIndex = 0;  
gameMode = 0;  
replayState = 0;

button = noteArray[arrayIndex]

no

increment waitTime;

waitTime = 4

**timer**

no

yes

**button down**

yes

audio play noteArray[arrayIndex];  
increment arrayIndex;  
waitTime = 0;

audio play off;  
waitTime = 0;

**button up**

yes

audio play off;  
arrayLength = 0;

arrayIndex = arrayLength

yes

no

gameMode = 2;  
waitTime = 4;  
audio play raspberry

decrement waitTime;

waitTime = 0

no

yes

**timer**

**start**

***Figure 2 -- Simple-Simon State Diagram***

# Simple-Simon Schematic

The simple-simon schematic shown in Figure 3 is quite simple -- just:

* 4 LEDs (capable of sinking 3.3 volts without a current limiting resistor) on pins a1, a3, a5, and a7,
* 4 SPST switches on pins b1, b3, b6, and b8, and
* a small piezoelectric buzzer that can run on 3.3 volts on pin a4.

Note there is a bit of asymmetry in the a/b pin use below, just to make my switches fit along the edge of the board, but you can use basically any pins you want for the LEDs and switches, and just change the “dim” statements in the BASIC program below to match. (The buzzer, OTOH, has to be on one of a3 thru a8 -- I chose a4 -- only those pins have frequency output capability.)

USB

a1

Flea-Scope™

a8

a5

BNC

*buzzer*

3V3

GND

5V

b1

b8

b6

3V3

GND

5V

b3

a0

b0

a7

a3

a4

***Figure 3 -- Simple-Simon Schematic***

# Connecting to Flea-Scope™

To program Flea-Scope™, just connect it to your computer, tablet, or phone, and open this webpage:

<https://rtestardi.github.io/usbte/stickos-basic.html>

Then Click the "Connect" button as shown in Figure 4:

A screenshot of a computer

Description automatically generated  
Figure 4

And select your Flea-Scope in the resulting dialog and click "Connect" again as shown in Figure 5:

A screenshot of a computer

Description automatically generated  
Figure 5

You should be connected to the web-page terminal emulator as shown in Figure 6:

A screenshot of a computer

Description automatically generated

Figure 6

You are now ready to enter StickOS commands and/or BASIC program statements.

# The BASIC Control Program

Enter the following BASIC control program at the StickOS command prompt for simple-simon:

**10 dim notearray[30]**

**20 dim gamemode, replaystate, waittime, arraylength, arrayindex**

**30 dim last\_sw[4]**

**40 dim buzzer as pin a4 for frequency output**

**50 dim led[0] as pin a1 for digital output**

**60 dim led[1] as pin a3 for digital output**

**70 dim led[2] as pin a5 for digital output**

**80 dim led[3] as pin a7 for digital output**

**90 dim sw[0] as pin b1 for digital input debounced inverted**

**100 dim sw[1] as pin b3 for digital input debounced inverted**

**110 dim sw[2] as pin b6 for digital input debounced inverted**

**120 dim sw[3] as pin b8 for digital input debounced inverted**

**130 configure timer 1 for 500 ms**

**140 on timer 1 do gosub timer\_popped**

**150 on sw[0]!=last\_sw[0] do gosub sw\_changed 0, sw[0]**

**160 on sw[1]!=last\_sw[1] do gosub sw\_changed 1, sw[1]**

**170 on sw[2]!=last\_sw[2] do gosub sw\_changed 2, sw[2]**

**180 on sw[3]!=last\_sw[3] do gosub sw\_changed 3, sw[3]**

**190 led[0] = 0, led[1] = 0, led[2] = 0, led[3] = 0**

**200 buzzer = 0**

**210 arraylength = 0**

**220 gosub addnote**

**230 halt**

**240 sub timer\_popped**

**250 print "timer\_popped"**

**260 if gamemode==0 then**

**270 gosub sw\_changed notearray[arrayindex], !replaystate**

**280 if replaystate then**

**290 arrayindex = arrayindex+1**

**300 endif**

**310 replaystate = !replaystate**

**320 if arrayindex==arraylength then**

**330 gamemode = 1**

**340 arrayindex = 0**

**350 waittime = 0**

**360 endif**

**370 elseif gamemode==1 then**

**380 waittime = waittime+1**

**390 if waittime==5 then**

**400 gosub gameover**

**410 endif**

**420 elseif gamemode==2 then**

**430 gamemode = 0**

**440 elseif gamemode==3 then**

**450 buzzer = 100**

**460 waittime = waittime+1**

**470 if waittime==3 then**

**480 replaystate = 0**

**490 buzzer = 0**

**500 arrayindex = 0**

**510 arraylength = 0**

**520 gamemode = 0**

**530 gosub addnote**

**540 endif**

**550 endif**

**560 endsub**

**570 sub lightbuzz n, down**

**580 led[n] = down**

**590 buzzer = down\*(440+110\*n)**

**600 endsub**

**610 sub sw\_changed n, down**

**620 last\_sw[n] = sw[n]**

**630 print "sw\_changed", n, down**

**640 if down then**

**650 if gamemode==0 then**

**660 gosub lightbuzz n, down**

**670 else**

**680 if notearray[arrayindex]==n then**

**690 gosub lightbuzz n, down**

**700 arrayindex = arrayindex+1**

**710 waittime = 0**

**720 else**

**730 gosub gameover**

**740 endif**

**750 endif**

**760 elseif gamemode!=3 then**

**770 gosub lightbuzz n, down**

**780 waittime = 0**

**790 if arrayindex==arraylength then**

**800 gosub addnote**

**810 endif**

**820 endif**

**830 endsub**

**840 sub addnote**

**850 notearray[arraylength] = random&3**

**860 arraylength = arraylength+1**

**870 arrayindex = 0**

**880 gamemode = 2**

**890 replaystate = 0**

**900 endsub**

**910 sub gameover**

**920 print "gameover"**

**930 buzzer = 100**

**940 led[0] = 0, led[1] = 0, led[2] = 0, led[3] = 0**

**950 gamemode = 3**

**960 waittime = 0**

**970 endsub**

**980 end**

(You can copy the program to the clipboard and then use the "Paste" button on the web-page terminal emulator user interface to load all the lines to StickOS at once.)

Lines 10-30 declare RAM variables, described in the state machine above. Lines 40-120 declare “pin variables” which are bound to pins on the board -- variables bound for output will transfer their values to the pins, and variables bound for input will transfer their values from the pins. Lines 130-140 configure a timer to call the **timer\_popped** handler subroutine every 500ms. Lines 150-180 detect button presses and releases and call the ***sw\_changed*** handler subroutine when they happen. Lines 190-230 simply set the initial values for variables and pins and then add the first note to the challenge, and simply halt execution, allowing the timer and button press and release handler subroutines to manage the rest of the game (the program “main loop” does nothing, hence the “halt” statement at line 230)!

The subroutines do the rest of the work:

* **timer\_popped** -- this is called every 500ms and checks what game state we are in, and runs the appropriate code for each game state.
* **lightbuzz** -- this turns LEDs and the appropriate buzzer tone on or off.
* **sw\_changed** -- this responds to switches being pressed or released, and runs the appropriate code for each game state.
* **addnote** -- this adds a random note to the end of the challenge array, as the game continues!
* **gameover** -- this sounds the raspberry buzzer tone and prepares to restart the game!

To save the program to the flash filesystem, type:

**save**

To set the program to autorun on MCU power-up, type:

**autorun on**

To run the program, type:

**run**

To stop the program, type:

**<Ctrl-C>**

# Diagnostics

What if your program doesn’t work? What if you have a bug in your buzzer or switch or LED circuitry? StickOS supports fully interactive control of the MCU. You can start running the program by typing “run” and then press <Ctrl-C> to stop it after all pin variables have been configured, and you will see:

> **run**

**<Ctrl-C>**

STOP at line 230!

At that point you can test all your pin variables! Let’s start by turning the buzzer on to 1000 Hz:

> **buzzer = 1000**>

You should hear the buzzer at 1000 Hz! Now let’s set to to 2000 Hz:

> **buzzer = 2000**>

Finally let’s turn it off:

> **buzzer = 0**>

Now let’s turn the LED’s (led[0] on pin a1 thru led[3] on pin a7) on and off, one at a time -- watch them change after each line:

> **led[0] = 1**> **led[0] = 0**

> **led[1] = 1**

> **led[1] = 0**

> **led[2] = 1**

> **led[2] = 0**

> **led[3] = 1**

> **led[3] = 0**

>

Now we can test the switches! We can print the pin variable values when all switches are released with:

> **print sw[0], sw[1], sw[2], sw[3]**

0 0 0 0

>

Now press a switch and repeat that command (you can use the “up arrow” button on your keyboard to recall previous commands), and you should see one of the switches reports “1” rather than “0”!

Note that these switches were declared (or “dimensioned”, in BASIC-speak) like:

**dim sw[0] as pin b1 for digital input debounced inverted**

This means they are automatically debounced by StickOS and they are also automatically inverted, so when there is a “0 volt” signal on the pin, the BASIC program will report a value of “1”, and when there is a “3.3 volt” signal on the pin, the BASIC program will report a value of “0”. StickOS automatically enables weak pull-up resistors for digital input pins.

# Conclusion: “You don’t even need an ‘app’ for that!!!”

Using StickOS BASIC inside your microcontroller, you can reprogram your microcontroller directly *using only a web-page terminal emulator and high-level BASIC algorithmic statements to manipulate the microcontroller (MCU) pins and peripherals*.

You can trivially create embedded systems to do useful work, like controlling a toaster oven for SMT reflow, or even make games, like simple-simon!

Even better, using just JavaScript and WebUSB and/or Web Serial in a Chromium-based web browser, you can write a *web-page terminal emulator* just once, and deploy it across most USB host computers -- a Windows, Mac, or Linux PC, a Chromebook, or even an Android phone!

“You don’t even need an ‘app’ for that!!!”

What could be easier???

# About the Author

A close-up of a person smiling

Description automatically generatedRichard Testardi has a wife and 17yo daughter and lives in Colorado. He is grateful most of the time and Christian. He loves anything outdoors or math/science related. In the future, he hopes to be teaching high school students. He lives without a cell phone (well, except a sim-less phone for interoperability testing!)!