The Projects of PROTOTYPE THIS

With Joe Grand

avrsimon: A Do-It-Yourself Game Kit

Prototype This, an engineering entertainment program on Discovery Channel, offered a view into the real-life process of designing and building unique prototypes. In this 13 episode series which was filmed over the course of 18 months and aired starting in October 2008, we set out to tackle the problems of today by creating crazy, one-of-a-kind inventions of tomorrow. I was the team's electrical engineer and hardware hacker, and shared the screen with Zoz Brooks, a roboticist and software designer specializing in human-machine interaction; Mike North, a material scientist and mechanical engineer; and Terry Sandin, a special effects veteran. We were challenged to build things that had never been done before, looked cool on TV, and could be completed within the extremely tight financial and time constraints of television production. It was a fantastic adventure and great experience to say the least!

This quarterly series of articles will cover the electronic aspects of some of my favorite projects from the show. My hope is that you will be inspired, learn something new, or use my work as a building block for your own open source project. Let's begin!

(All photos and figures by Joe Grand.)

Launched in 1978 by Milton Bradley, the addictive, flying saucer-shaped memory game of Simon remains an icon of early electronic games. The premise is simple: Repeat the sequence of lights and tones. This article presents my Simon clone running on an Atmel ATtiny2313, an eight-bit AVR microcontroller (**Figure 1**).

My implementation of the game combines a number of basic microcontroller functions such as reading switch inputs and turning LEDs on and off, with more complicated ones such as using sleep modes to extend battery life and playing sounds. I've also designed in a few gameplay enhancements such as no sound mode, fast mode, no LED mode, and reverse mode. Details of the original Simon design can be found in US Patent #4,207,087 entitled "Microcomputer controlled game" (**www.google.com/patents/about?id=MAIyAAAAEBAJ&dq=4207087**).

Full engineering documentation (including source code and PCB Gerber plots) is available on my website (**www.grandideastudio.com/portfolio/avrsimon**/). If you don't want to build the circuit from scratch, a kit is available from Parallax (**www.parallax.com**) and includes all of the necessary components, including a pre-programmed microcontroller and custom circuit board. The only additional items you'll need to build the kit are a soldering iron, solder, wire snips, and a battery.

Hardware

Like the original game, avrsimon's user interface is simple, comprising of four buttons, four LEDs, and a piezo

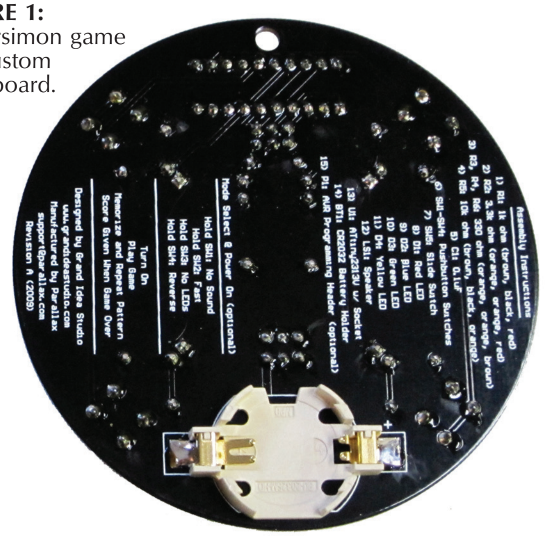

FIGURE 1: Assembled avrsimon game with custom circuit board.

buzzer for playing sounds. Refer to the schematic (**Figure 2**) and bill of materials (**Table 1**) for specific component information.

The Atmel ATtiny2313 microcontroller (**www.atmel.com/dyn/products/product_card.asp?part_id=4660**) is an eight-bit microcontroller based on the AVR enhanced RISC (Reduced Instruction Set Computer) architecture. It features 2 KB of Flash, 128 bytes of in-system programmable EEPROM, 128 bytes of RAM, up to 20 MHz clock speed, 1.8V to 5.5V operation, and up to 18 general-purpose I/O pins all in a 20-pin package. Peripherals include two timers, four PWM channels, analog comparator, and a Universal Serial Interface/UART. It is a nice, low-cost part (around $2 in small quantities) suitable for many embedded systems projects.

One side of each button (SW1, SW2, SW3, SW4) is connected to Port B 1, 4, 0, and 3, respectively. The other side of the buttons are connected to ground. The internal pull-up resistor (with a value between 20K and 50K ohm, according to the ATtiny2313 datasheet) is enabled on each of the Port B pins, removing the need for four external resistors. The buttons are active low, so the microprocessor normally sees a high signal ('1') — due to the pull-up resistor — when the button isn't being pressed. When a button is pressed, the microprocessor sees a low signal (0).

The cathodes of the four LEDs matching the four colors of the original Simon game (D1 red, D2 blue, D3 green, D4 yellow), are connected to Port D 0, 3, 1, and 2, respectively. A current-limiting resistor (R1, R3, R4, R2) connected to VCC is used in series with each LED to limit the amount of current allowed to flow through it which sets the brightness and prevents excessive current from damaging the LED. Like the buttons, the LEDs are active low and set up in a current sink configuration; meaning they will turn on when the output signal on one of the Port D pins is low (0). When we want to turn an LED off, we simply set the corresponding pin's output value to high (1). Each port pin on the ATtiny2313 can safely sink 20 mA which is well above what the LEDs on avrsimon require. When the red LED is on, it requires 1.3 mA, the green 3.0 mA, blue 0.13 mA, and yellow 3.2 mA.

One side of the piezo buzzer, LS1, is connected to ground and the other side is connected to Port B 2 via R6 — a current-limiting resistor. Instead of using the general-purpose I/O function as with the button inputs and LEDs, Port B 2 is used as its special function — Output Compare for Timer 0 — which outputs waveforms that will drive the piezo buzzer. Turning the game on and off is achieved with a simple slide switch used in an SPST configuration that connects and disconnects the battery supply from the

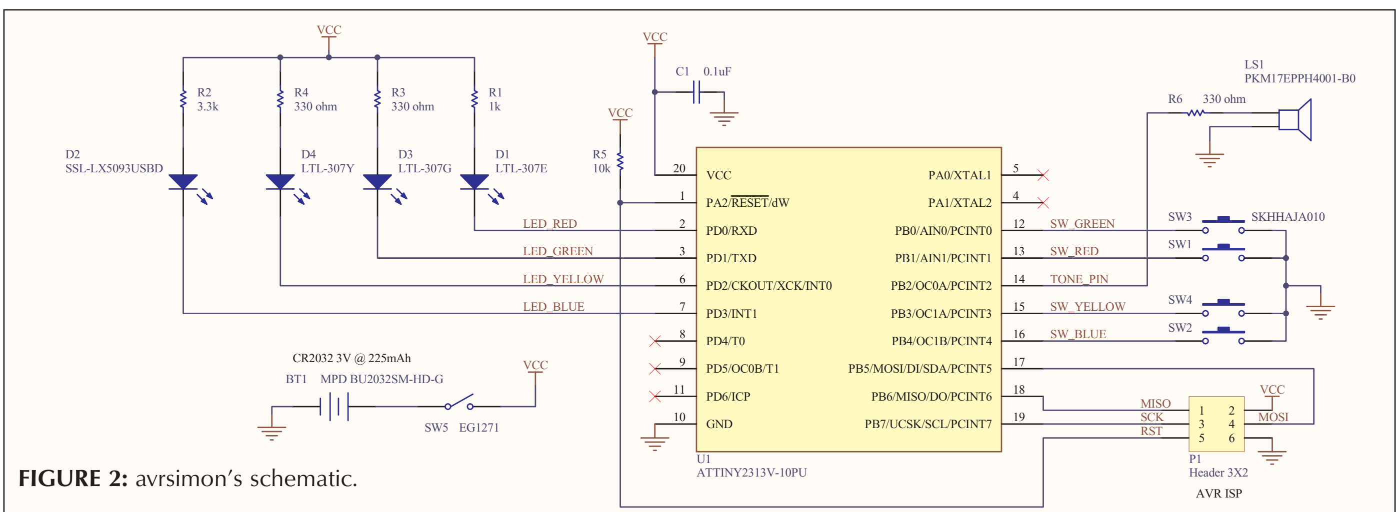

FIGURE 2: avrsimon's schematic.

VCC bus of the circuitry. The system is powered with a single CR2032 3V lithium coin cell which is easy to obtain from any local drugstore, convenience store, or electronics outlet. The CR2032 has a very nice current capacity for its size (20 mm in diameter) of approximately 225 mAh, although the lithium battery chemistry works best for applications requiring very low current discharge (tenths of mA) over months or years of use. Its maximum recommended continuous discharge is 3 mA which is what avrsimon draws during gameplay. When the game is not being used and while the system is waiting for a button press to begin a new game, U1 is placed into a sleep mode and current consumption is reduced to a scant 19.5 uA. With typical gameplay of a few hours a day, a single battery should last over a month.

There are a few other discrete components used in this design: C1 is a standard bypass/decoupling capacitor connected close to the VCC input of U1. R5 is a pull-up resistor connected to the active-low /RESET line of U1 that keeps the microcontroller operating properly (e.g., not resetting) unless the pin is intentionally pulled low. P1 is the standard six-pin AVR In-System Programming (ISP) header. This is an optional part that is only required if you plan on making changes to the firmware and want to reprogram U1 while it is in-circuit.

Firmware

At the highest level, the operation of avrsimon's firmware is straightforward:

Upon power-up, the hardware is initialized within the aptly named *hardware_init()* function which brings the system into a known state. The function configures the I/O pins (LED pins as outputs, switch pins as inputs), sets up the timers (Timer 0 is used for tone generation and Timer 1 for timeout counting used during gameplay), and enables the Pin Change Interrupt.

Then, *simon_config()* is called which sets the gameplay mode based on the combination of pushbuttons SW1-SW4 held down during power-up. If no buttons are pressed during power-up, then the game will play in the normal mode. Other modes include no sound mode, fast mode, no LED mode, and reverse mode, and serve as an additional challenge for advanced users. Details of each special mode are discussed in the How to Play section of this article. After all configurations are complete, we move into the core *simon_game()* routine. Immediately after entering *simon_game()*, the system is configured to enter a low-power sleep mode and to awake on a button press from any of the four buttons. Sleep mode is attained by calling a specific sequence of functions/macros which define the type of sleep mode we want to enter, configure the interrupts, and then go to sleep:

```
// prepare to go to sleep/idle mode...
set_sleep_mode(SLEEP_MODE_PWR_DOWN);
cli();        // disable global interrupts
sleep_enable();
sei();        // enable global interrupts
sleep_cpu();  // now go to sleep to save power
```

Sleep mode conserves a significant amount of power by shutting down all unused modules of the microcontroller and only keeping the absolute essential features awake. In our selected Power-Down Mode (SLEEP_MODE_PWR_DOWN), all clocks and oscillators are disabled, all peripheral modules are turned off, and the only ways to awaken the device are via specific resets (external, watchdog, or brown-out), serial interface or INT0 external interrupts, or a Pin Change Interrupt generated on specific port pins. For more details on sleep mode implementations, see **www.nongnu.org/avr-libc/user-manual/group__avr__sleep.html**.

With avrsimon, when a button press is detected via the Pin Change Interrupt on Port B 0, 1, 3, or 4 (corresponding to SW3, SW1, SW4, SW2, respectively), the system springs to life and begins the game.

The game itself is comprised of two core functions: *simon_play_moves()* and *simon_read_moves()*.

simon_play_moves() first pulls a number from *rand()* — a linear feedback shift register used as a pseudo-random number generator (**http://en.wikipedia.org/wiki/PRNG**) — which is seeded at the beginning of each game with the current value of U1's Timer 1 counter and whatever

Qty	Reference	Description	Distributor/Part #
1	BT1	Battery holder, CR2032 Lithium coin cell, SMT	Digi-Key, BU2032SM-HD-GCT-ND; **www.digikey.com**
1	C1	0.1 µ, 50V 10% bypass capacitor ceramic axial, X7R	Digi-Key, 1109PHCT-ND
1	D1	5 mm LED, red high-efficiency diffused, 19 mcd, 2.0V	Digi-Key, 160-1705-ND
1	D2	5 mm LED, blue diffused, 300 mcd, 3.5V	Digi-Key, 67-1751-ND
1	D3	5 mm LED, green diffused, 19 mcd, 2.1V	Digi-Key, 160-1706-ND
1	D4	5 mm LED, yellow diffused, 29 mcd, 2.1V	Digi-Key, 160-1703-ND
1	LS1	Piezoelectric buzzer, 72 dB @ 25V pp, 4 kHz, PCB mount	Digi-Key, 490-4694-ND
1	P1 (optional)	Header, 3x2 vertical, male, PCB mount	Digi-Key, 3M9459-ND
1	R1	1.0K ohm, 5% 1/4W	Digi-Key, 1.0KQBKND
1	R2	3.3K ohm, 5%, 1/4W	Digi-Key, 3.3KQBK-ND
3	R3, R4, R6	330 ohm, 5%, 1/4W	Digi-Key, 330QBK-ND
1	R5	10K ohm, 5%, 1/4W	Digi-Key, 10KQBK-ND
4	SW1, SW2, SW3, SW4	SPST momentary pushbutton switch, 100 gf, PCB mount	Mouser, 688-SKHHAJ; **www.mouser.com**
1	SW5	SPDT slide switch, 300 mA, PCB mount, L = 2 mm	Digi-Key, EG1918-ND
1	U1	AVR microcontroller, DIP20	Mouser, 556-ATTINY2313V10PU
1	U1b (optional)	Socket, DIP20	Digi-Key, 3M5465-ND
1	PCB	Printed circuit board	Parallax

TABLE 1: avrsimon's bill of materials.

pushbutton was pushed to start the game. The number from *rand()* is limited from 0 to 3 (corresponding to one of the four possible LED colors on avrsimon) and then stored in U1's internal EEPROM — a non-volatile storage container with individual byte addressing — at the memory address equal to the current length of the sequence. The game then plays the entire sequence that is currently in its EEPROM, contiguously from address 0 until the end of the sequence is reached. For more details on EEPROM handling, see **www.nongnu.org/avr-libc/user-manual/group__avr__eeprom.html**.

The game then enters the *simon_read_moves()* routine which waits for the player to begin replaying the sequence. For each move in the sequence, the corresponding address of the EEPROM is read and compared with the player's input. If the values do not match, then the player must have pushed the wrong button and the game will end via the *simon_failed_input()* function. If the values do match, then the player pushed the correct button and the game proceeds to the next move in the sequence. If the player successfully repeats the entire sequence, the game jumps back to *simon_play_moves()* to add another move to the sequence and repeats the process until the game is over.

Sound

Tone generation is based on the xyloduino project (**www.rocketnumbernine.com/2009/03/27/xyloduino-simple-arduinopiezo-organ/**) and modified to support arrays of octaves, notes, and durations in order to create melodies. Timer 0 — a hardware peripheral internal to U1 — is used as an eight-bit counter that will toggle the OC0A output pin (Port B pin 2) from low to high or high to low when the counter value (TCNT0) matches the value programmed into the Output Compare Register (OCR0A). The toggling of the output pin will generate a square wave at the desired frequency that is fed into the piezo buzzer LS1. The *play_note()* routine is passed the octave, note, and length of the sound. It configures the Timer/Counter Control Register (TCCR0B) and OCR0A accordingly, waits for the sound to be played, and then disables the counter:

```
if (note)
{
    TCCR0B = pre[(int)octave];
      // set the prescaler depending
      // on what  octave is selected
    OCR0A = note >> (octave%2);
      // there are two octaves for each
      // prescale setting, so adjust
      // accordingly
}
delay_ms(length);
     // play sound for specified length
TCCR0B = 0;
     // turn off sound
```

In order to play a melody, a sequence of notes stored in an array must be played one after the next:

```
for (i = 0; i < STARTGAME_SND_SIZE
* 4; i = i + 4)
{
    // octave, note, length
    play_note(pgm_read_byte(startgame_
    snd_p + i),
        pgm_read_byte(startgame_snd_p
        + i + 1),
        pgm_read_word(startgame_snd_p + i + 2));
    delay_ms(50); // pause in between notes
}
```

The four sounds generated on the original Simon were based on four primary notes of a bugle which sound "in tune" when played in any order. avrsimon closely mimics those notes (**www.waitingforfriday.com/index.php/Reverse_engineering_an_MB_Electronic_Simon_game**):

Tone 1: Blue, 392 Hz (G note)
Tone 2: Yellow, 330 Hz (E note)
Tone 3: Red, 262 Hz (C note)
Tone 4: Green, 196 Hz (G note)

Development Environment

avrsimon was developed on OS X using CrossPack for AVR (**www.obdev.at/products/crosspack/download-de.html**). Formerly known as AVR MacPack, the package contains the core compiler, debugger, and AVR-specific

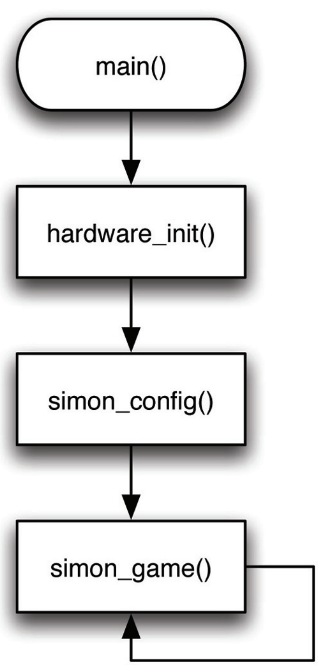

FIGURE 3: High-level program flow.

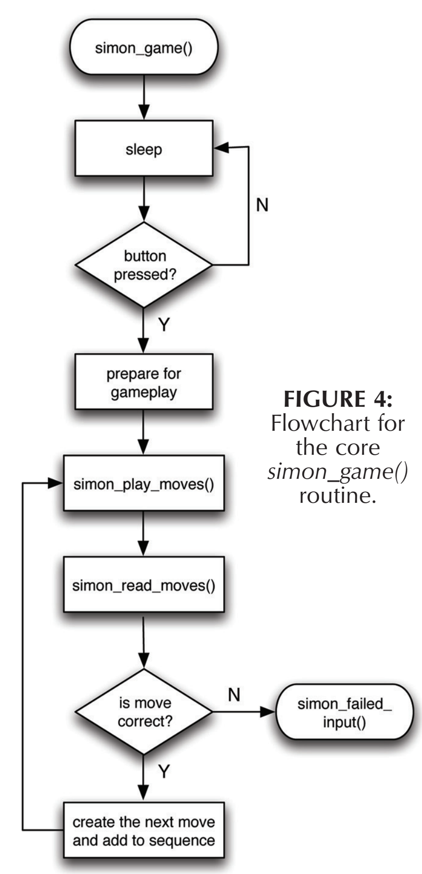

FIGURE 4: Flowchart for the core *simon_game()* routine.

tools, and integrates seamlessly with Apple's Xcode.

In-circuit device programming was achieved using an adafruit industries USBtinyISP interface (**www.adafruit.com/index.php?main_page=product_info&cPath=16&products_id=46**).

If you choose to modify avrsimon's firmware, you'll need to recompile the code and reprogram it into the microcontroller. To reprogram the microcontroller, hook up the USBtinyISP to P1 (insert and solder this optional six-pin male header onto avrsimon if you haven't done so already), locating pin 1 by its square pad on the backside of the circuit board. Then, open a Terminal window and go to your /firmware/ directory. Finally, run the *make install* command which launches the avrdude application twice with different parameters: once to load the compiled Hex file into Flash memory and once to set the device's configuration fuses (defined in the Makefile):

1) avrdude -c usbtiny -p attiny2313 -U flash:w:main.hex:i
2) avrdude -c usbtiny -p attiny2313 -U hfuse:w:0xd1:m -U lfuse:w:0xe4:m

For more information on Atmel AVR development, see:

- AVR Freaks web page, **www.avrfreaks.net**
- adafruit industries' AVR Tutorial web page, **www.ladyada.net/learn/avr/**

How to Play

In a nutshell, avrsimon gameplay is as follows:

- Turn On
- Play Game
- Memorize and Repeat Pattern
- Score Given When Game Over

When you first power up the game, a start-up tune will welcome you. Press any of the pushbuttons to start playing. The game will generate a sequence of lights and sounds that you are to repeat, starting first with a single element. After the sequence has been presented, simply press the button(s) corresponding to the LED that was illuminated and repeat the pattern. The sequence length will increment each time you successfully repeat the pattern, making the game increasingly more difficult as you go. The maximum sequence length is 255.

When your game is over, a short tune will be played, the correct LED in the sequence that you were supposed to have entered will be illuminated, and your score will be given by a series of blinking LEDs. The green LED corresponds to hundred; the red LED to ten; and the blue LED to one. For example, if you failed after a 13 element sequence, the red LED will first blink once and then the blue LED will blink three times.

avrsimon has a few optional twists to make gameplay more fun and interesting for advanced players. These special modes are selected by holding down one of the pushbuttons SW1-SW4 while first turning on the game (multiple pushbuttons can be held down at one time to create various combinations):

- **SW1: No Sound/Quiet Mode**

No sounds are generated while in this mode, making it perfect for gameplay late at night, in a library, conference session, or classroom.

- **SW2: Fast Mode**

This mode increases the speed of which the LED sequence is played and reduces the length of time allowed for you to repeat the sequence. Normally, you have five seconds to make a decision for each move. In fast mode, you're only allowed two.

- **SW3: No LED Mode**

No LEDs are illuminated while in this mode. You'll have to repeat the pattern based on sound alone.

- **SW4: Reverse Move**

This mode operates like a FIFO (First In, First Out) stack in which you have to replay the sequence in reverse order (instead of repeating the exact order that the game presents).

To revert back to game's normal mode of operation, simply power cycle the unit. Enjoy the game! **NV**

Joe Grand is an electrical engineer, hardware hacker, and president of Grand Idea Studio, Inc. (**www.grandideastudio.com**), where he specializes in the invention, design, and licensing of consumer products and modules for electronics hobbyists. He can be reached at joe@grandideastudio.com.