SOLIS

Arduino Driven - LED Lamp & TFT Touchscreen Interface

Andrew Nieuwsma | 2018-04-22 | SENG 5831 | Final Project

Prepared for: Dr. Amy Larson, College of Science and Engineering

University of Minnesota

Introduction

Solis, the Latin word for light, is an Arduino driven lamp with touch screen interface. Solis is a stand-alone product and does not require any external device to control or manage it. I became interested in developing a lamp after seeing a demonstration of NeoPixels, the RGB LEDs that provide the lamp its light. Around the same time, I remember reading about the human circadian rhythm, and how light can affect our body's natural wake/sleep cycle. The project I demonstrate today is the 2nd version of Solis. I chose to re-develop Solis because I was unsatisfied with the result of the project from several years ago.

The original prototype for Solis was also a physical lamp that was Arduino driven, but the original design used a LCD screen and 5 tactile buttons. However, the implementation of my first version had to be very scaled back because of memory, performance & timing problems I encountered in development. The embodiment of version one abandoned the display and the features that relied on it and used a potentiometer to control brightness and a button to control color.

In taking SENG 5831 and learning about different ways that tasks could be periodically scheduled I was curious if I could revive the concept of the lamp and make it something more useful. While I did rely on my memory of the original project, I did not use any previous code base or designs in this project, it is all 'fresh' and of my own composition (aside from the scheduling algorithm and some code snippets from a vendor- see References for a list).

Table of Contents

Learning Objective and Outcomes

My primary learning objectives were centered around resolving the pain points I recall from version 1 of Solis. The largest pain point was that scheduling in version 1 relied on the main loop only to schedule and call functions. The loop was 'in order execution' with a very ugly nest of branches and conditionals. There was certainly no ability to 'schedule' a task to run within a predetermined period. The second largest pain

point was the interface. Tactile buttons often need software debouncing (unless using special hardware to square out the signal), and debouncing usually is in the order of milliseconds which complicates the overall timing requirements of the software. In addition, the user interface was a LCD screen with 2 lines of 30 characters worth of text, so I was extremely limited in what messages I could send to the user and how they would interact with it.

In this project I applied what I have learned in SENG 5831 about rate monotonic, non-preemptive scheduling to implement a fair scheduler that would service tasks according to a pre-determined period. In addition, I also learned a lot about defining algorithms that were the least intrusive to the 'critical section' of the system and how to manage state in the user interface.

The level of abstraction for this project is medium. I used the driver libraries provided by the hardware vendors to control the LEDs, read and set timers, and read and write from the touch screen. My primary educational focus was on learning how to manage the resources of the system, so I purposefully ignored communication protocols and details as much as possible.

Challenges

In implementing the project, I ran into numerous challenges that I will describe in detail.

Protecting the Critical Section

The critical section refers to a shared resource that multiple actors depend upon to complete their task. In this project the most important critical section was the state of the LEDs. The LED API did not come with a mechanism to determine the state of the LEDs. Instead the burden of responsibility was on the user to remember if they had turned the LEDs on or off, as well as what color and brightness level had been specified.

Remembering LED state was very important because there were multiple actors that could set the state of the LEDs. The ON/OFF switch could set the LEDs on or off, the Brightness button could cycle a brightness level, the color buttons could change the color of the LEDs, and the LED Timer could turn the LEDs off once the timer had expired.

Early in the development cycle, several actors (functions in the code) could turn on or off the LEDs which proved very problematic, because it meant that state could get switched without other actors being made aware of it. Instead a single scheduled function ("manageLEDs") was created that would implement the requested state change to the LEDs (On/Off, Brightness, Color). A flag variable was created that would indicated if the LEDs were On or Off, with the only function allowed to modify the state of variable being the "manageLEDs" method. In addition, another flag variable was set to indicate that an intent to switch state had been requested. Many actors could set this variable to true, but only the "manageLEDs" function could set to false, as "manageLEDs" was the actor responsible for servicing the request. Finally, an intended led state flag was created that allowed other actors to request the LED be turned off or on, this variable gave the "manageLEDs" function the insight into what requested state had been most recently requested.

Creating Timers and Measuring Expiration

Solis relies on several timers to determine when key functions of the system should occur. One timer determines when the LEDs should automatically switch off, and another timer determines when the

touchscreen backlight should go off. The system uses a real-time clock that can give the wall-time to second precision, as in 60 seconds in a minute.

The state for the timers can be, OFF (i.e. no expiration) or can set to a value (an enumeration) in the range of minutes to hours. When a timer is set, because the corresponding front end interface has been 'pressed', the system will register the current time + the expiration length as the future time that the timer should expire (in addition to changing the state of the user interface to represent the new expiration state). The main scheduler will periodically execute the "manage'X'Timer" function which will check if the time has expired and register state changes of the system as appropriate.

By partitioning the timer functions out to individual methods that were controlled by individual tasks, the system complexity was reduced and helped guarantee that the timings of the system tasks would not be incumbered by continually increasing functionality (and code overhead) to a few main functions. This was a learned experience from the previous implementation of Solis, as continually expanding the scope of functions led to hard to diagnose problems, partially due to shared resource problems, and partially due to code complexity.

Managing the Interface

The user interface of the system is a small TFT touch screen that could map to 240x320 pixels. In experimenting the author found that the smallest reasonable size target to try to press by finger without mistaking it for another target was about 40x40 pixels. The compressed space restricted the amount of functionally to around 48 different parts of the screen. In practice however, it made more sense to further restrict the parts of the screen down to 17 elements: 12 for color selection @ 40x40, 1 for brightness @ 80x120, 1 for On/Off @ 80x120, 1 for LED timer @ 80x120, 1 for backlight timer @ 80x120 and 1 for a special user function @ 80x240. These sizes made it reasonably trivial to hit the intended section of screen with high precision.

The elements of the screen are mapped to UI blocks (through XY coordinates), that invoke 'back end' operations to manage state. Originally the code to determine what element had been pressed, and then manage state, and update the UI was all in one massive function. However, as the code evolved it made increasingly more sense to make the UI code as small as possible, making it only responsible for mapping a press of an element to what top level function should be called, which then managed state. This made the code much simpler to follow and diagnose when problems occurred. The code had been developed so that if new screen size was to be implemented, only some definitions of boundaries would need to be updated allowing the UI to scale without problem.

Debugging

Debugging an embedded software project is always difficult. Arduino makes it a little less painful with its integrated support of UART and a built-in console that can connect over USB. However, unlike non-embedded application environments that can usually attach debuggers and step through the code, there is not functionally that does it conveniently. Debugging at times, as is common with much software development, is very crucial to resolving system critical issues, usually caused by the programmer mistyping, or mis understanding the system.

A common pattern when debugging Arduino, or other platforms for that matter, is to use print statements to express code state or flow of control. However, as Arduino is setup in a constant while loop, the output

can be very verbose. Additionally, serial printing is very time consuming (given the 16MHz processor) so time sensitive code paths can be dramatically impacted.

The two largest complications of using print statements to debug Arduino is getting the right level of verbosity (without having to tweak the code constantly) and limiting the memory overhead of the debug messages. In this application the author developed a debugging verbosity structure and function that would register a message at a certain verbosity level. If the system was in a greater than or equal to verbosity level, the message would be printed out when executed, else it would be omitted. This solved the first problem, but not the memory management issue.

The Arduino Uno has 32k of flash memory (program space), and 2k RAM. When RAM or flash memory gets to around 70% full, performance warnings start to be output as part of compilation. In the authors experience, memory performance issues are very serious in the embedded space, as they are usually tied to system instability. Normal performance issues general imply that code may execute slower, however from the authors experience, memory performance generally means that the system will experience buffer or stack overflows, at random times, and make the system very fragile. The author did not come up with a long-term solution but did implement compile time flags that would not include debug code unless debug mode was specified. It is recommended that a production system not be compiled with debug flags enabled.

Clean Coding

Arduino as a platform is a hybrid of C/C++. It uses parts of the STL, but the code wont easily port to a different microprocessor, say TI, or PIC. Because of time constraints the author focused more on the functionality of the code then the development environment, which meant that the entire code base was in a single file (1,300 lines long) and was developed using the Arduino IDE. The implication of that decision is that there is a lot of code to bounce around in, and its pretty easy to get disoriented.

To help remedy the potential coding problem, the author paid special attention to refactoring and making the code as clean as possible. Coding best practices were applied such as using version control as often as possible to keep a record of the progression of changes. Every function also was given a comment block describing the parameters and functionality of the method. The top most part of the code, where “#include”s and variable declarations are created was kept very clean and organized, separated by comment blocks, so that it should be fairly easy to split the code into multiple files or classes.

Product

Major Features

1. Turn the LEDs On or Off
2. Select the Color of the LEDs (Red, Orange, Yellow, GreenYellow, Green, Cyan, Blue, Maroon, Purple, Magenta, Pink, White)
3. Adjust the Brightness of the LEDs (20%-100% in steps of 20%)
4. Cycle a timer for LEDs to auto turn off (NOTimer, 5s, 30s, 15m, 30m, 60m, 90m)
5. Cycle a timer for the backlight to auto turn off (NOTimer, 10s, 30s, 1m, 2m, 5m)
    a. Turn off the Backlight
6. Taste The Rainbow – calls a function that makes the strand look like a shimmering rainbow for 5 seconds.

Hardware Components

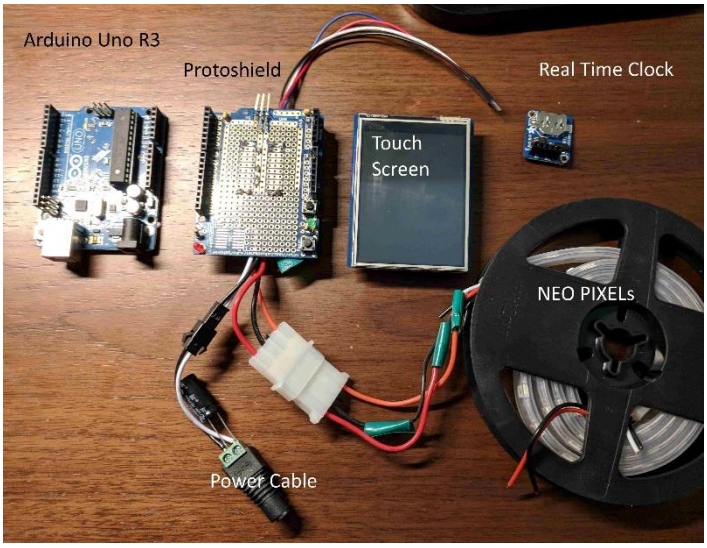
The major components of the system include:

1. Arduino Uno R3 (Revision 3 is needed to support the TFT)
2. DS1307 – Real time clock
3. TFT Touch Screen
4. NeoPixels strand – NeoPixels are individually addressable RGB LEDs
5. Arduino Protoshield – used to stack the TFT on top of the Arduino while exposing pins for power, RTC
6. 5v power supply – with a 100 Pico farad capacitor in parallel to protect the LEDs

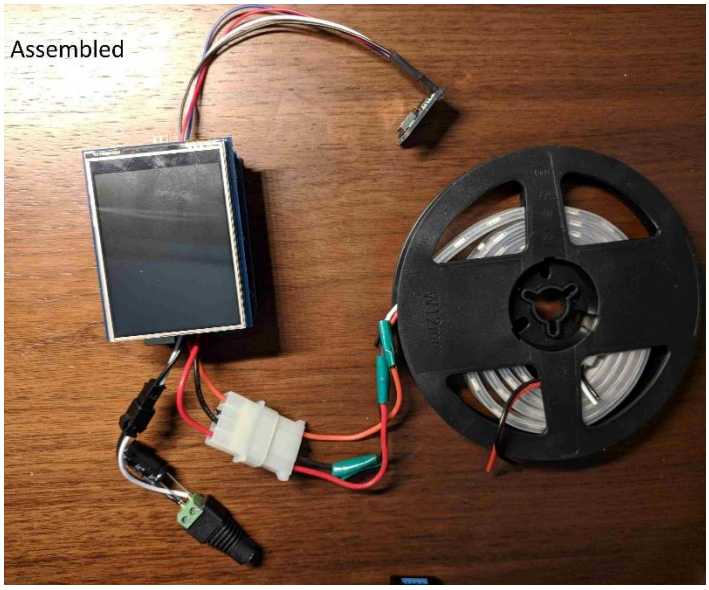
In the current, 'non-polished' version of the hardware, I also used a variety of Molex connectors and wire to make my product as condensed and modular as possible, as I wanted to avoid continually using a bread board once I verified the hardware would work.

Hardware Layout

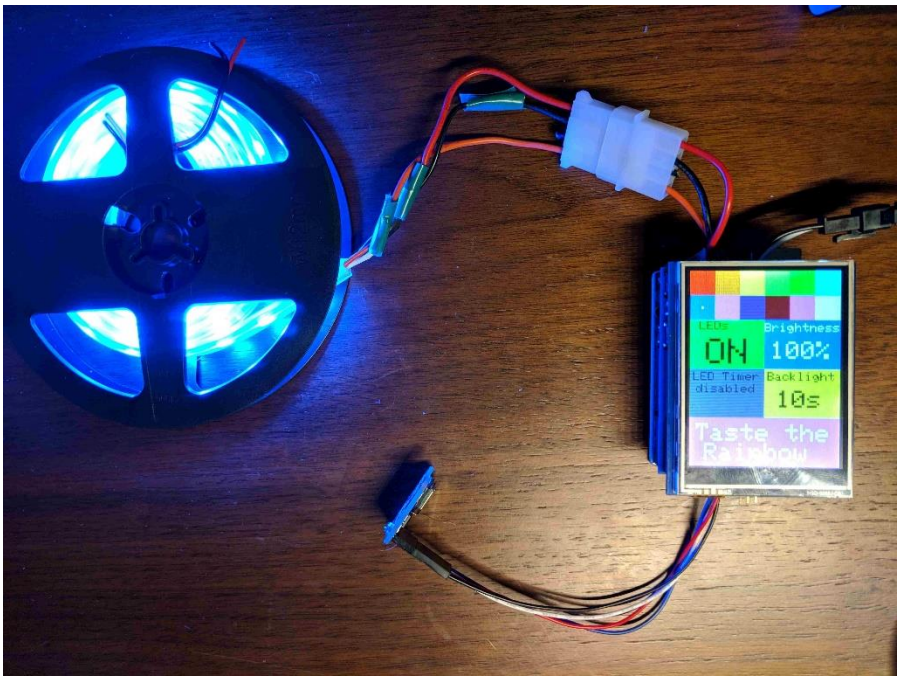
Exploded



Assembled



Demonstration



Screens

Here is the main screen, approximately to scale.

Scheduled Tasks

Function	**Description**	**Priority**	**Period**(ms)
manageLEDS	Turns LEDs On or OFF	2	100
userInterface	Interprets screen presses, calls functions to respond	1	75

manageBacklightTimer	Determines if the backlight should be off or on	6	200
manageLEDTimer	Determines if the LEDs should be off or on	7	300
paintLEDTimer	Paints the screen with the timer value	8	5000

The scheduled tasks invoke all other necessary tasks (See API in APPENDIX).

Future Work

While the current code base is stable, there is potential for future work. The future work can be categorized as functional opportunities and operational opportunities.

Operational opportunities modify how the operation of the system works, but do not add or remove features from the system. Such operational opportunities should include examining memory usage and optimizing allocation. The current compiled software uses 67% of program storage space (32k) and 72% of dynamic memory (2k) with debug flags disabled. From the author's experience, just a few more dozen lines of code will probably take the system past the 70% mark where memory performance becomes an issue.

Future work could also be done to look at storing non-changing values (like print messages to the touch screen) into EEPROM to conserve program space and dynamic memory. Additionally an operational opportunity exists to increase the precision of touch screen debouncing, which links a single intended press of the touch screen to a single event in the system.

Functional opportunities add or enhance the functionality of the system. Currently the system has a single main screen, the system could be expanded to create multiple 'screens' that the UI would display. These additional screens would add features like: display the system time, edit the system time, set an alarm for the future (where the LEDs will turn on at a predetermined time). However, the memory restrictions will likely prohibit adding new features until something is done to give the system more memory.

References

https://www.space.com/18917-astronauts-insomnia-light-bulbs.html

https://learn.adafruit.com/adafruit-neopixel-uberguide/basic-connections

https://learn.adafruit.com/adafruit-2-8-tft-touch-shield-v2

https://learn.adafruit.com/ds1307-real-time-clock-breakout-board-kit

APPENDIX

API

Name	**Description**
setup	Runs once at boot and initializes the system
loop	The main while loop of the system, runs forever
writeDebugMessage(debug_verbosity msgVerbosity, String msg)	Prints a debug message if the verbosity is high enough
pickColor(TS_Point p)	picks the color and draws the selection box
userInterface	this is the main UI method, it registers touches on the touch screen and then decided what action to take
spawn_all_tasks	spawns all tasks
spawn(int (*fp) (), int id, int p, int priority)	spawn a task
setLEDResource(bool stat)	sets the flag of intent to switch state
paintScreenBlack	paints the whole tft black
paintColors	paints the 12 color boxes to the screen
initColors	fills the colors array with color values
manageLEDS	sets the state of the LEDs
paintTheRainbow	paints the taste the rainbow button
paintLEDTimer	paints the timer box, and will change what it looks like based on the state of system
cycleBacklightTimerSettings	cycles through the backlight timer enum
cycleLedTimerSettings	cycles through the LED timer enum
paintBacklightTimer	paints the backlight timer button with values
paintClock	prints the month - day, hour : minute to the screen
cycleBrightnessSettings	cycles through the brightness enum
paintBrightnessButton	paints the brightness button, VERY static
paintLEDStatusON	updates the led on / off button with ON value
paintLEDStatusOFF	updates the led on / off button with OFF value
setBacklight(bool stat)	turn the backlight off or on
isBacklightTimerExpired	calls the update timer expired function and returns state
updateBacklightTimerExpired	if the timer watchdog is off, then the timer is NOT expired else if time > timer then it is expired
setupBacklightPin	backlight is on pin 3
resetBacklightTimer	reset the backlight timer to be NOW + watchdog length, will be invoked on every touch screen press
manageBacklightTimer	if the time is expired, then turn off the backlight, else turn it on
isLEDTimerExpired	calls the update timer expired function and returns state
updateLEDTimerExpired	if the timer watchdog is off, then the timer is NOT expired else if time > timer then it is expired
resetLEDTimer	resets the LED timer

isLEDON	returns if the LED is on, based on the value of CURRENT_PIXEL_STATE
manageLEDTimer	if the time is expired, then turn off the LED, else turn it on
printTasksToConsole	print the tasks to the serial console
printTimeToConsole	prints the time to the serial console
tasteTheRainbow	invokes theaterChaseRainbow(0)
colorWipe(uint32_t c, uint8_t wait)	Fill the dots one after the other with a color
theaterChaseRainbow(uint8_t wait)	Theatre-style crawling lights with rainbow effect