EEC 172 Lab 1

Ziyi Chen
University of California, Davis
Davis, United States
czych@ucdavis.edu

Ayden Leu
University of California, Davis
Davis, United States
ajleu@ucdavis.edu

ACM Reference Format:
Ziyi Chen and Ayden Leu. 2026. EEC 172 Lab 1. In . ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

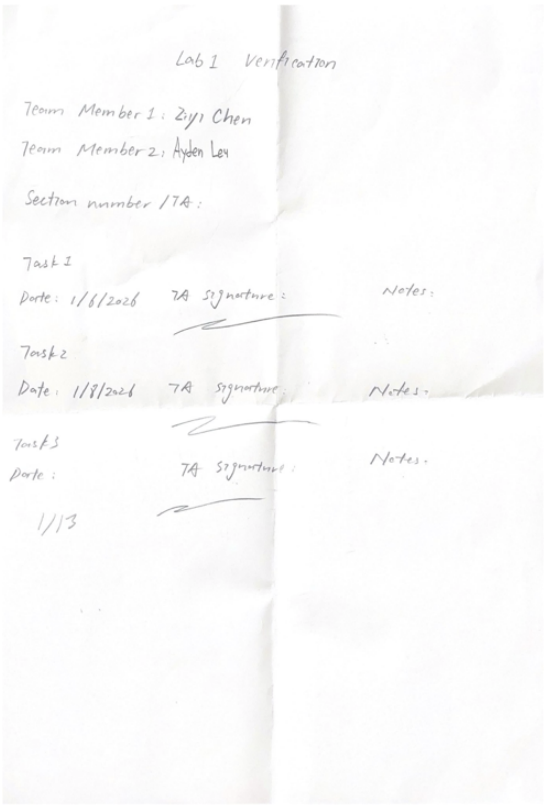

Figure 1: Completed Lab 1 checkoff sheet verified during lab checkoff.

1 Introduction

This lab covers the fundamental software development workflow for the SimpleLink CC3200 microcontroller. In Part I, example projects from the CC3200 SDK were imported into Code Composer Studio, built, and run on the CC3200 LaunchPad with the CCS debug environment. A simple modification to the code was made to change the rate at which the LED blinks, allowing the build and debug workflow to be verified. Part II focused on creating a custom application that communicates with GPIO and UART peripherals. The TI SysConfig tool was used to set up pin multiplexing for LEDs, switches, UART communication, and an external GPIO pin, while application logic was developed to control LED patterns based on switch inputs and display status messages via a serial console. In Part III, the completed application was programmed into the CC3200's on-board serial flash memory with CCS UniFlash, allowing the program to run without the debugger.

2 Background

2.1 Hardware and Software Tools

This lab utilized the **CC3200 LaunchPad** as the primary embedded hardware platform. Software development was conducted using **Code Composer Studio (CCS)**, which provides an integrated environment for writing, compiling, and debugging C programs. **CCS UniFlash** was used to program compiled firmware into the CC3200's non-volatile flash memory.

2.2 Compiling for Embedded Systems

The lab emphasized core embedded software development techniques. C source files were compiled into object files and linked with precompiled static libraries provided by the CC3200 SDK to generate an executable. This executable was then converted into a binary image suitable for flashing onto the device. The lab also highlighted the distinction between loading programs into RAM for debugging and flashing firmware into non-volatile memory for persistent execution.

3 Goals

3.1 Part 1: Testing Examples

The goal of this part was to run an example CC3200 project in Code Composer Studio, successfully build and debug the program, and make a simple modification to change the LED blinking behavior. By completing this task, we learn the basic embedded development workflow using CCS, including project importation, compilation, and loading firmware onto the CC3200 for debugging.

3.2 Part 2: Application Program Exercise

The goal of this part was to modify a template program that interfaces with an external console window, and two switches and LEDs on the CC3200 LaunchPad. The modifications included changing the initial console message, and allowing interaction with the on-board LEDs with the on-board switches. By completing this task,

Conference'17, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Signal Name	Component	Pin Number	IO Mode
GPIO_09	Red LED	64	Output
GPIO_10	Yellow LED	1	Output
GPIO_11	Green LED	2	Output
GPIO_13	Switch 3 (SW3)	4	Input
GPIO_22	Switch 2 (SW2)	15	Input
GPIO_28	Pin 18 (P18)	18	Output

Table 1: Pin Configuration Settings

Component	GPIO Base	GPIO Bit (GPIO_PIN_)
Red LED	A1	1
Yellow LED	A1	2
Green LED	A1	3
Switch 3 (SW3)	A1	5
Switch 2 (SW2)	A2	6
Pin 18 (P18)	A3	4

Table 2: Component GPIO Settings

we learned how to configure the CC3200 LaunchPad's pins, interact with the CC3200 LaunchPad using an external computer, and how to write software that utilizes the CC3200 LaunchPad's hardware.

3.3 Part 3: Programming Flash Memory

The goal of this part was to flash the program we created in Part 2 onto the CC3200 LaunchPad. By completing this task, we learned how to flash a program onto the CC3200 LaunchPad such that the program will remain in non-volatile memory through power cycles.

4 Methods

4.1 Part 1: Testing Examples

We imported an example project named `blinky` from the CC3200 SDK into CCS and built it following the part 1 lab instructions. The program was built to ensure there were no errors, then loaded into the CC3200's RAM via CCS's debug mode. Execution was then resumed to ensure that the onboard LEDs blink as expected. To satisfy the checkoff requirement, we increased the LED blinking rate ten times faster than the default implementation. Following the modification, the project was rebuilt and reloaded onto the CC3200. The updated program was run and observed to ensure that the LEDs were blinking at the increased rate.

4.2 Part 2: Application Program Exercise

4.2.1 Pin Configuration. First, we created a pin multiplexing configuration using the TI SysConfig Tool to allow us to interface with the on-board components using GPIO signals. The GPIO signals found in Table-1 were configured and exported to `pin_mux_config.c` and `pin_mux_config.h` to be used with our program.

4.2.2 Setup. We then imported an example project named `uart_demo` from the CC3200 SDK into CCS and built on it. The initial console message was updated to match the one found in the lab instructions.

4.2.3 "Global" Variables. Next, two variables are initialized for each switch we want to use, one for the switch state on the current frame and one for the switch state on the previous frame (e.g `switch2IsPressed` and `switch2WasPressed`). Each clock cycle, a switch's corresponding `WasPressed` variable is updated to the switch's `IsPressed` variable, and the `IsPressed` variable is updated to match the switch's current state using `GPIOPinRead(...)`. The parameters needed for this function are the same as the first two parameters of `GPIOPinWrite()`, which will be covered later. If a switch is pressed on this frame and was not pressed on the previous frame, that means the user just pressed the switch.

4.2.4 Functionality. To meet the requirements laid out in the lab instructions, pressing switch 3 will put the program into `LED_BINARY` mode and lower pin 18's output to zero. Pressing switch 2 will put the program into `LED_BLINK` mode and raise pin 18's output to one. We also implemented an additional `IDLE` mode where all of the LEDs turn off, which can be enabled by pressing both switch 3 and switch 2 at the same time. To detect when a switch is pressed, we use the previously defined `IsPressed` and `WasPressed` variables for each switch to update a new `currentMode` variable to the corresponding state. Each state is represented by a unique unsigned short.

4.2.5 Updating Components' State. To change any board component's state, the `PIOPinWrite(...)` function is used, with the first parameter being the component's "base," the second being the component's pin number, and the third either being zero or the component's pin number depending on if we want to turn it off or on, respectively. To determine a component's base and pin number, we referenced Table 5-14 in the CC3200's tech reference manual. The name of the variable that corresponds to a board component's base is a combination of the component's GPIO Module Instance name and "`_BASE`." The corresponding GPIO bit can also be found in Table 5-14 of the CC3200's tech reference manual. For convenience, this information has been compiled into Table-2.

4.2.6 Global Counter. To control the timing of certain tasks, a global counter variable is created and incremented every clock cycle. We can then modulo this counter value against certain value and execute an `if` branch when it equals zero (`if(counter % value == 0){...}`). We can't use the built-in delay functions found in example projects like `blinky` as it would stall the program until the delay finishes.

4.2.7 LED BINARY Mode. When the program is in `LED_BINARY` mode, we need to remember the previous binary counter state. To achieve this, we created a global unsigned short (`hardwareState`) and manipulate its bits to match the state of each board component we use (e.g. red LED is on, the right-most bit gets set to 1). We cannot read the current state of components like the three LEDs as their IO mode is set to `Output` and cannot be set to both `Input` and `Output`. With the hardware state kept track of, we then use the hardware state of the three LEDs as the binary counter and increment it every `100000` clock cycles.

4.2.8 LED BLINK Mode. When the program is in `LED_BLINK` mode, we set and keep each LED's state on for `100000` clock cycles and off for `100000` clock cycles.

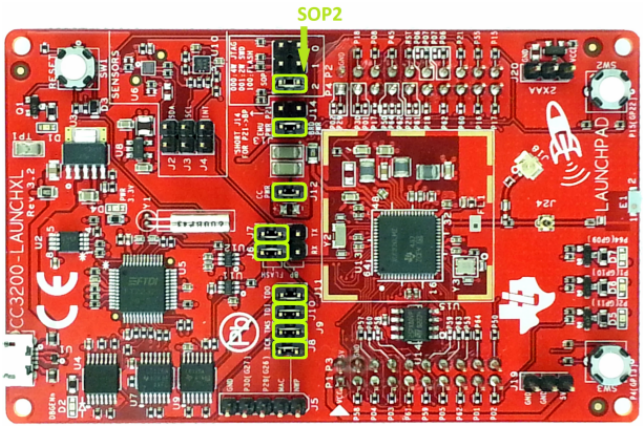

Figure 1: CC3200 Board w/ Jumper Configuration

4.2.9 IDLE Mode. When the program is in `IDLE` mode, we just set each LED's state to off.

4.3 Part 3: Programming Flash Memory

We opened CCS UniFlash and followed the lab instructions. We verified the board was connected to our development PC, determined its COM port, and selected the `\sys\mcuimg.bin` option under the System Files section. We then selected the `.bin` file that was generated from our build and debugging process, which was named `lab1-part2_uart_demo.bin` and found under the `Debug` folder in `lab1-part2_uart_demo`.

Before flashing the `.bin` file, we made sure a jumper was placed on `SOP2` as shown in the Figure-1. Once verified, we programmed the board with our program. To verify it was flashed correctly, we replugged the board into our development computer, opened a PuTTY terminal to connect to the board's console, and pressed the board's reset button to see the initialization message. We also verified the program's functionality, we interacted with both switches.

5 Discussion

The main challenges we faced during this lab were getting familiar with Code Composer Studio, determining which GPIO base and pin corresponded to the board's components, finding a way to work around the lack of parallelization, and flashing the program. We became familiar with Code Composer Studio by repeating the steps laid out in the lab instructions. We determined each board component's GPIO base and pin by digging through the CC3200's documentation. We worked around the lack of parallelization by using a global counter that increments up each clock cycle. When it came to flashing the program to the CC3200 board, the application failed to load into the CC3200's serial flash memory. Despite verifying the correct COM port and binary file selection, the flashing process did not succeed. The issue was resolved after identifying that the SOP2 jumper had not been installed before connecting the LaunchPad to the computer. Once the SOP2 jumpers were properly setup, UniFlash was able to successfully program the flash memory, and the application ran properly after power cycling.

6 Contributions

Both team members worked collaboratively throughout all parts of the lab. We worked together to set up the development environment, including configuring debugging tools in Code Composer Studio. During Part II, both members contributed to configuring pin multiplexing using the TI SysConfig tool, implementing GPIO and UART functionality, and developing the application logic for switch polling, LED control, and console output. The ability to switch modes, put the board in an idle state, working around the lack of parallelization, and final project organization was done by Ayden. Debugging and verification of program behavior were performed together through pair programming and joint testing. In Part III, both team members participated in programming the application into the CC3200's serial flash memory using CCS UniFlash, troubleshooting flashing issues, and verifying correct standalone execution after power cycling.

Overall, the lab was completed through close collaboration, shared debugging efforts, and collective verification of all checkoff requirements.