GSoC 2021 Final Report

apertus° Association

AXIOM Remote: Firmware improvement and extension

Aman Singh

Mentors: Sebastian Pichelhofer, Andrej Balyschew

Sub Mentor : Priya Pandya

August 22, 2021

CONTENT

Overview

My work involves advancing and extending the firmware of AXIOM Remote. During the GSoC coding period, I have been involved in the tasks mentioned below:

- Adding transition animations in the UI of AXIOM Remote
- Add a text input GUI
- Set up a timer to measure performance of the remote.

Because of some hurdles along the way, I have not been able to finish, but I am currently working on tasks and in the preceding time, I will get them merged.

My organization has supported me and has helped in every way to speed up things. They have sent me AXIOM Remote hardware and helped me set it up. While setting up the hardware, the FT4323H mini-module was not working, so we had to debug and order a new one, which took a lot of time. But after setting the hardware I have been able to finish as much as possible and because of the hardware, I can work on the PIC32 MCU properly and enjoy the process.

In upcoming sections, I have explained my work and the code accompanying it.

Work Product

All the code produced by me during GSoC 2021 can be found in this repository. Branch.

About AXIOM Remote

AXIOM Remote is the remote control for AXIOM beta. It's hardware includes PIC32MZ, two PIC16s and a LCD for the menus and options, with different buttons and rotor. The software runs "bare metal" and has no graphic acceleration, so each pixel is drawn manually in the software. Firmware of AXIOM Remote was initially prototyped in C and is being ported to C++.

Components

- PIC32MZ as the core processor 2 small PIC16s are used for the handling push buttons, rotor and the LED I/0 operations.
- 2.8'' 320x240 TFT (ILI9341) from Adafruit is used as the display.
- USB-C Connector is used.

Task 1 - Re-add Transition Animation

The C firmware (old firmware prototype version) had transition animations in which the entire content of the screen moves or wipes out of the frame as per animation speed (4 orientations can be selected). This was done by the second framebuffer of the entire LCD. At the start of the transition, a copy of the screenshot was essentially made in the specified framebuffer, and then with the next upcoming frames, the contents of that second framebuffer was moved or cropped in the direction of movement.

***Note** : screen refers to the entire content of the LCD visible at one time (AXIOM Remote UI guidelines).*

My work basically involved re-adding transition animation in current C++ firmware.
In terms of displaying the framebuffer on screen, firmware currently redraws the framebuffer constantly. It clears the LCD, fills the framebuffer again, and transfers the data to LCD constantly.

Initial Approach

It is an implementation of the previous approach of C firmware, as described in the above paragraph.
In current C++ firmware, the class objects needed to modify, for the above approach were **painter, menuSystem and ILI9341DisplayDriver.**

- **Painter -** It is responsible for populating the framebuffer with the content that needs to be displayed on screen.
- **MenuSystem -** It is responsible for switching screens when the user changes screen.
- **ILI9341DisplayDriver -** It is responsible for displaying framebuffer on TFT display and all the functions related to the display of AXIOM Remote.

Key changes

1. Introduced a second framebuffer in AXIOM Remote firmware's linker script and in the AXIOM Remote Firmware Visualizer.
2. Modified the menuSystem, so when transition needs to happen, it would change the screen, only after the current content of framebuffer has been copied to the second framebuffer.
3. Modified painter, so when transition is triggered, it captures (copies) the framebuffer and saves its content into a second framebuffer.
4. Modified ILI9341DisplayDriver, to handle two framebuffers and show transition animation.
5. For displaying the framebuffer and handling transition animation in Visualizer, a separate object would be defined called **VirtualLCDDevice** (mimics the actual display driver of AXIOM Remote firmware in the Visualizer).

***Note** : AXIOM Remote Visualizer is a tool to emulate the actual code running on the PIC32 and pixels displayed on the 320x240 LCD on a PC.*

Limitation with Initial Approach

- Whenever a transition is triggered, firmware copies and captures the framebuffer's content into another similar second framebuffer. All this can be avoided, making transition animation less heavy on CPU (as explained and done in Next Approach)
- Requires painter to copy content of framebuffer to another framebuffer, and menu system to handle when transition occurs, which violates the Single Use Principle of Object Oriented Programming.

Current Approach

For transition animation to happen, a second framebuffer is required that stores the content of the screen that needs to be displayed. The key difference between current approach and initial approach is the way in which the second framebuffer is filled.

As we know firmware redraws everything constantly, that is, it fills the framebuffer and displays it, constantly. Let's call the framebuffer that is being currently used for this process as, **active framebuffer,** the framebuffer that is not being displayed lets call it **passive framebuffer**

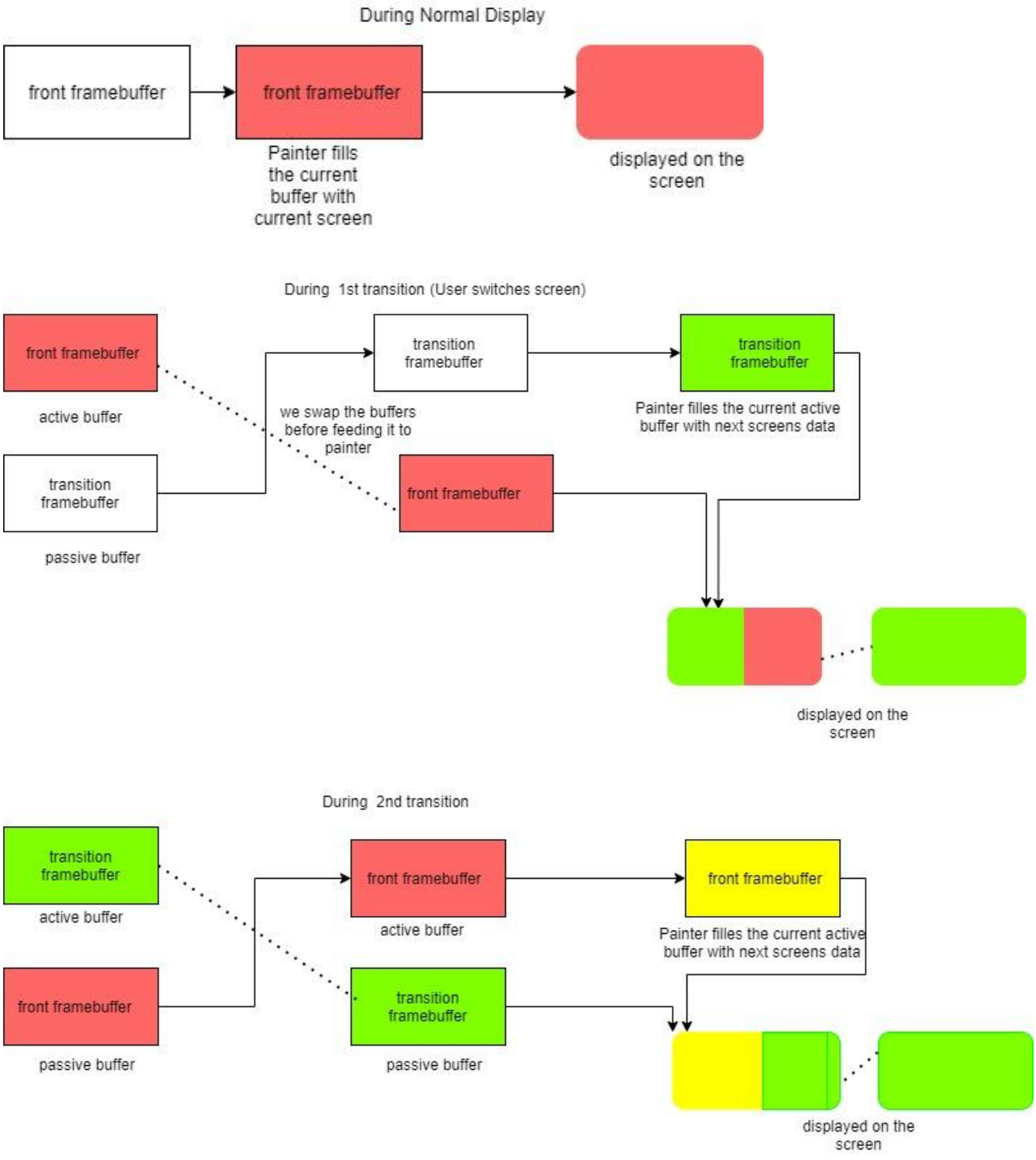
In the current approach, at the time of when transition is triggered, instead of copying the content of the currently displayed framebuffer, let's call it **front framebuffer,** into a second framebuffer, lets call it **back framebuffer**, we can just swap the **front framebuffer** (that is provided to the painter for drawing) with **back framebuffer**.

By swapping the framebuffer, our **back framebuffer** becomes our **active framebuffer** and is filled with the content of the screen that needs to be displayed, and our **front framebuffer** will have content of the screen that was just displayed, and is present on screen.

Event	Active framebuffer	Passive framebuffer
First Drawing	**front framebuffer**	**back framebuffer**
Drawing just before 1st transition	**back framebuffer**	**front framebuffer**
Drawing just before 2st transition	**front framebuffer**	**back framebuffer**

This swapping happens every time we change screen on AXIOM Remote.

During normal operation, **the active framebuffer** is populated and displayed constantly by firmware, like it is happening currently.



Challenges Faced

- I was initially designing code based on AXIOM Remote Firmware visualizer, because of which most of the code I wrote was not optimized to run on embedded systems. So, it used to crash,

when run on AXIOM Remote hardware. I have corrected this. My mentor Sebastian had sent AXIOM Remote hardware, which speeds up testing and trying out new things on Remote.
- While designing buffer swapping in the current approach, it was hard for me to visualize and swap the buffers asynchronously, so drawing is not interrupted on LCD and also framebuffers are swapped, when needed.

Link of video of transition animation on AXIOM Remote - Link
Modifications in codebase for Various animations - Link
Modifications in codebase needed for buffer swap while changing screen - Link
Link to complete codebase Repository - Link

Task 2 - Text input GUI

The Task requires extending firmware with a Text Input GUI. After having discussion with my mentors about how we should take input from the user, it was decided to go with a QWERTY style keyboard.

Components in Text Input GUI

- A text field to show what user is typing
- A keyboard that can switch its layout between Numeric, Uppercase and Lowercase characters based on the user's need.
- Basic keys to control functionalities like going to previous screen,

Screenshots of text input screen

AXIOM Remote firmware has components that are common between different screens. These components are called widgets in firmware, and are placed in a separate widget object class.

So, the keyboard used in the Input GUI was also technically a widget, so I have made it into a widget that can be used on any screen.

***Note** : screen refers to the entire content of the LCD visible at one time (AXIOM Remote UI guidelines).*

Challenges Faced

- Similar to transitions, I had written code based on AXIOM Remote Firmware Visualizer, which was not optimized for embedded systems.
- Designing the basic operations of the keyboard like traversing along the keyboard handling corner case, button events were a bit tough, but mentors helped overcoming it.

Link to Video of Text Input GUI on AXIOM Remote - Link
Modifications in Codebase for keyboard Widget - Link
Link to complete codebase Repository - Link

Task 3 - Setup timer in PIC32

Task requires setting up the timer of PIC32, for measuring performance and timing functions in AXIOM Remote firmware. PIC32 offers five timers, with a 16 bit period register (which can be combined to give a 32 bit timer) and a core timer with a 32 bit period register.

Using Timer to measure performance in firmware

For timing functions

We save the timer to a register or memory location at the beginning of the section, we want to time and subtract this value from the timer value reached at the end of the section.

For checking average frame rate

A timer would generate a periodic timer interrupt at 1Hz (i.e. every second), so we can check and see the number of rendered on the LCD per second, that is frames per second.

Using Core Timer

Initially, It was decided to implement the core timer of PIC32, to accommodate various ways we can measure performance in firmware.

About Core Timer

Core timer is implemented in the form of two co-processor registers: the ***Count register***, and the ***Compare register***. The ***Count register*** is incremented every two system clock (SYSCLK) cycles. The **Compare register** is used to cause a timer interrupt. An interrupt is generated when the Compare register matches the Count register.

SYSCLK frequency in AXIOM Remote - 192 MHz
Core timer uses a 32 bit register (max 4,294,967,295) and counts up at SYSCLK/2, that is our core timer will tick for every 10.41 ns, and will go up to 257.6976 sec. So, for 1 sec timer interrupt, we would need a compare register to be set to 96000000.

Core timer is visible through Coprocessor 0, or CP0.
Register no. 9 - Count Register (Count Processor cycle count)
Register no. 11 - Compare Register (core timer interrupt controller)

We can access the registers in CP0 using coprocessor instructions, such as mfcx (move from coprocessor x register) and mtcx (move to coprocessor x register).

For example, System software accesses the registers in CP0 using coprocessor instructions such as mfc0 (move from coprocessor 0 register) and mtc0 (move to coprocessor 0 register). The following below two lines, clears the core timer counter register and initializes its period register with the value in a0:

```
mtc0    zero, Count
mtc0    a0, Compare
```

But this is now actually not needed since the Microchip compiler has built-in pre-defined functions for the same.

```
_CP0_SET_COUNT(0)
_CP0_SET_COMPARE(a0)
```

Limitations of Using Core Timer

The difference between core timer and normal timer module is that, core timer's ***Count register*** *doesn't reset to 0, after it matches with* **Compare register**, *so, we have to use delta method i.e., difference between*

different values of count register to measure time taken and for average frame rate measurements we have handled, it in the Interrupt Service Request of the core timer.

So, using just timer 1 would be better, where we do not have such overheads, where we do not need to follow additional steps that makes using the core timer a bit more heavy on CPU compared to when using other timers.

Using Timer 1

Timers in PIC32 are controlled by Peripheral bus clock 3, and our system clock runs at 192 Mhz. We set the Peripheral bus clock 3's frequency to SYSCLK freq/12 = 16 Mhz, allowing us to get 1 second interrupts with a period value of 62500 (in the range of 16 bit timer) .

Link to Timer function - Link

Challenges Faced

- Setting up the timer required understanding some core concepts of PIC32 that I was lacking. I had to understand and research a lot about things like PRISS register, CPO register, oscillators in PIC32, timer module in PIC32. At the end, i was able to understand and carry on with the task.

Task 4 - Documenting the Process of setting up the AXIOM Remote

Organization had sent me AXIOM Remote hardware. I had updated the apertus wiki with documentation about the process of setting up the AXIOM Remote hardware for development purposes, and flashing the firmware in PIC32 and PIC16s.

Link to AXIOM Remote development notes - Link

Post GSoC Plans

At the beginning of January, I had talked about various tasks available with AXIOM Remote, and I wasn't able to do much because I didn’t have any prior experience in that field. Since then, I have come a long way. I now understand PIC32 structure and embedded systems. I have AXIOM Remote hardware that speeds up the development and learning process quite a lot.

I had many hurdles along the way during the GSoC period. I will finish all my GSoC tasks properly and get them merged. I am continuing working on AXIOM Remote, and would like to be a sub mentor next year, as my mentors have helped me with many things, this was my first ever professional experience, and I have grown a lot.