Computer Science Capstone - Journal 1

Ty Gillespie

November 18, 2023

Goals:

1. Install and configure the PlatformIO CLI, as well as CMake and a few other C++ build tools. Completed.
2. Initialize the project. Completed.
3. Bootstrap the GitHub repository. Completed.
4. Fix any build errors with submodules and make the build scripts platform-agnostic by the end of Tuesday. Completed.
5. Begin writing speech handler class by the end of Thursday. Diverted.
    - This class just caused include path problems, and Espeak's API is actually nice. I wrapped Espeak init into a function, and now it works quite nicely.
    - I'll have to wrap it eventually for asynchronous speech and interrupts, but an entire class was overkill.
6. Map all the buttons to actual functions. Completed.
7. Store the current equation and history. Completed.
8. Implement a function that converts mathematical symbols into English equivalents. Completed.
9. Fix loads of driver problems and upgrade development environment. Completed.
10. Convert the character speaking function to use a map of strings to chars. Completed.
11. Implement basic expression evaluation using Tinyexpr. Completed.

My Research and What I Learned: *Speech_handler: Sometimes the Existing APIs are Good Enough* I have previously used the Espeak-NG API on Windows NT and Windows CE 5.0. I (naively) assumed that the Arduino wrapper would work in a very similar way. However, I was extremely wrong.

The Arduino library provides a super convenient method to synchronously output a string of text (something no other Espeak implementation does to my knowledge), declared like so:

```
void espeak::say(const char* text);
```

This approach has a few problems, like it only accepting constant character pointers (const char*s) (easily solved by using the .c_str() method of std::string), and the fact that it speaks synchronously (meaning that any code on the same thread as espeak, (in this case the main thread), isn't able to execute while speaking). As such, I may eventually have to dig down into the guts of the Espeak API again and deal with callbacks and all the other fun things. Alternatively, I could just make a generic speak function, something like:

```
void speak(const std::string& text);
```

then have overloads for character pointers (in the rare case we need them) and other such types. If I went with this approach, I'd most likely have the code create another thread in setup() that looks for a global flag when it's time to speak. There's still a problem with this, but it'll be easily solved, especially because I've already used the Espeak API on embedded hardware, even if it was a different architecture. No matter what I decide to go with though, it'll most assuredly be better than my original approach, which was to create a C++ class to wrap all of this up into a convenient interface. It sounds good as a base, but it caused me endless amounts of problems:

1. The Espeak constructor requires a reference to an AnalogAudioStream. This is fine in most cases, except when you're trying to use Espeak as the type of a class property (unfortunately for me, exactly what I was trying to do). Because of the way C++ constructors work I had to declare the AnalogAudioStream as a property of the class too and pass a pseudoreference to it, which only worked sometimes, on some compilers.

2. The espeak.say() function was complaining when I threw C++ strings at it, even when I converted them to char pointers with the .c_str() method.

3. When attempting to include the required header files, the compiler complained about double definitions for reasons that still elude me to this day, mainly because this is the exact thing that header guards are supposed to protect against, and the files I was including most certainly had them.

4. And so on...

Needless to say, I abandoned that idea fairly quickly, opting instead to use the extremely simplified and nice Espeak API directly.

The Calcvox Website The Calcvox website is actually extremely simple when you dig down into it. It has absolutely no JavaScript and requires no cookies, and not even a good internet connection, to load. This means I can even use it in things like Edbrowse (a command line-based text editor and web browser

for the blind that doesn't really support JavaScript). The domain registrar is Cloudflare, and the site is also hosted on Cloudflare pages, constantly checking a GitHub repository for new changes and updating the content accordingly. The modern web is super overcomplicated, and as such, JavaScript is basically required these days. It allows you to have interactive websites and even do things as simple as have the same navbar show up on all pages of your website. You can also handle this (using something like PHP), but we don't get that kind of control with Cloudflare pages. I didn't want to hardcode all the items because we're constantly adding and removing them and thought "there has to be a better way." It turns out that "better" is subjective, but I did end up with something that works, a Python script that inserts templates into HTML, like so:

```
import argparse
from pathlib import Path
from jinja2 import Environment, FileSystemLoader

def render_html(filename, templates):
env = Environment(loader=FileSystemLoader(filename.parent))
template = env.get_template(filename.name)
context = {}
for template_path in templates:
template_name = template_path.name
with open(template_path, "r") as template_file:
context[template_name] = template_file.read()
rendered_html = template.render(context)
with open(Path("dist") / filename, "w") as output_file:
output_file.write(rendered_html)

def main():
Path("dist").mkdir(exist_ok=True)
parser = argparse.ArgumentParser(description="Render HTML templates using Jinja2")
parser.add_argument("-i", "--input", required=True, type=Path, help="Input HTML file")
parser.add_argument("-t", "--template", action="append", required=True, type=Path, help=
args = parser.parse_args()
input_path = args.input
template_paths = args.template
render_html(input_path, template_paths)
print(f"Rendered {input_path}")

if __name__ == "__main__":
main()
```

This is laughably simple honestly; all it does is take a list of templates on the command line and a file to render, inserts the templates where necessary, and hands you a file. However, it's also a massive hack that requires Python to run, as well as some packages to be installed. It's now integrated into the website's

build process though, so it's extremely seamless once you get the environment set up. Another problem we had with the website was attempting to display our timeline. Originally, we tried using an iframe and just embedding the Google Sheet directly, but that had issues with screen readers. It would be the worst kind of ironic for a website for the blind to have accessibility problems, so I went back to the drawing board. I eventually came up with a super interesting hack. Basically, I generated a link to our Google Sheet as CSV (comma-separated values). I then wrote a tiny script to convert this into a nice little HTML table, like so:

```
import pandas as pd
import requests
import argparse
from pathlib import Path

def fetch_csv(url, output_path):
r = requests.get(url)
r.raise_for_status()
csv = r.text
output_path = Path(output_path)
# Really ugly kludge just to make it work.
csv = csv.replace("NaN", "")
output_path.write_text(csv)

def csv_to_html(csv_path, output_path):
csv_frame = pd.read_csv(csv_path)
csv_frame = csv_frame.fillna("")
csv_frame.to_html(output_path, index=False)

if __name__ == "__main__":
parser = argparse.ArgumentParser(description="Convert a Google Sheets CSV to an HTML tabl
parser.add_argument("url", help="URL to the sheet")
parser.add_argument("--output-path", "-o", help="What to save the HTML file as")
args = parser.parse_args()
url = args.url
output_path = args.output_path or "sheet.html"
fetch_csv(url, "sheet.csv")
csv_to_html("sheet.csv", output_path)
Path("sheet.csv").unlink()
print("Done")
```

This works almost flawlessly! There are some very minor problems with column headers in some instances for reasons I absolutely do not understand yet, but they're pretty minor, so I haven't investigated it yet.

Upgrading the Development Environment When I first tried the Arduino IDE, I found it to be completely inaccessible to screen readers. I then looked into how to use the Arduino CLI, but very quickly opted to switch to PlatformIO.

This was for a few reasons, but the biggest thing was the improved support for the board we're using. I didn't know it at the time, but PlatformIO also has a Visual Studio Code extension. By the time I discovered it had one though, I had already gotten into the flow of things using the PlatformIO CLI, so saw no real reason to change. This very quickly changed when I found that I all of a sudden couldn't deploy code to the prototype anymore. I had very recently switched laptops, and the necessary drivers for deploying to the ESP just wouldn't work on my new one. I still never quite figured out why, but it ended up not mattering because after a week of frustration with it, I decided to switch to VS Code and never looked back.

Converting Mathematical Characters Into English While adding character echo to the firmware (i.e. making the calculator read what you're typing as you type it), I ran into a problem with Espeak. When sending a single char to it (like '+'), it would only say the raw character code. Char pointers don't have this problem, so any multi-character strings are fine. Converting the raw char to a char* would've been a lot of work, though. I would've needed to allocate two bytes (one for the character and one for the NULL terminator), strcpy() it, and it would've been super messy. I instead opted for a function to convert keycodes into speech, like so:

```
std::string convert_character(const char character) {
static const std::map<char, std::string> char_map = {
{'0', "0"},
{'1', "1"},
{'2', "2"},
{'3', "3"},
{'4', "4"},
{'5', "5"},
{'6', "6"},
{'7', "7"},
{'8', "8"},
{'9', "9"},
{'+', "plus"},
{'-', "minus"},
{'*', "times"},
{'/', "divide"},
{'.', "point"},
{'=', "equals"}
};
auto it = char_map.find(character);
if (it == char_map.end()) {
return "";
}
return it->second;
}
```

This works quite well! I did have a few iterations of this function though:

- The first one was a super messy implementation I hacked together just to make it work. I knew I'd eventually clean it up though, and a few days later, I did exactly that.
- My second implementation is extremely similar to what you see above, with only a few characters changed. The first line that starts with "static" didn't say that initially. Marking the variable with static basically means that the variable will only be allocated/created once. I ran into a problem with this function if I pressed buttons too quickly because it had to recreate the map and all the strings again. The ESP is fairly RAM-limited, so this proved to be a problem. So now, the map is only created once, the first time the user presses a key.

Accomplishments:

- Got fully working text-to-speech on the prototype.
- Successfully mapped all buttons to C++ functions.
- Implemented equation history, although this system will definitely be expanded later.
- Reimplemented and modernized the entire development environment, changing it from a fairly kludgy combination of Notepad 2, Python, Scoop, and the PlatformIO CLI on a Windows virtual machine to a fully integrated Visual Studio Code setup with the PlatformIO IDE extension.
- Set up PuTTY as a serial monitor to check what the ESP was doing while debugging.
- Implemented loads of software functionality, such as:
  - Character echo (i.e., each button press will generate a spoken message telling you what key you pressed, primarily so you don't have to proofread your equation every time you press a key).
  - A C++ function, `std::string convert_character(const char character);`, to convert mathematical symbols into their Plain-English equivalents. For example, the symbol "+" would become "plus," the symbol "-" would become "minus," etc. This is presently only really useful in the character echo logic, but I opted to make it a reusable function for things like rereading the equation later.

Progress on Timeline: I'm about where I expected myself to be on my timeline. I definitely did hit some unexpected roadblocks that caused me to lose time, like my laptop absolutely refusing to see any drivers I installed to deploy code to the prototype, making me have to rip my entire development environment, on two platforms, out and start all over again, multiple times, but in the early weeks, I gained a massive amount of time due to things not taking as long as I expected, in addition to having full and ready access to a prototype way ahead of schedule.