Firmware programming – a journey full of adventures

In this section of the guide to make your own Smart Glasses, I will start by showing the series of steps needed to have functioning Smart Glasses, explain the reasoning behind the choices that have been taken in programming the firmware, and mention the difficulties one might encounter programming such a firmware.

# Instructible – “running” the firmware

Once you have a function circuit with all the components mentioned in our hardware section, you will need to make these components interact one with another. This is the role of the onboard microcontroller, in our case – the ESP32. To do so, you will need to *flash* the firmware it: programming it with code you have written. If you simply want to get the glasses working, you can use the firmware we have programmed (and which we will go into detail in the following paragraphs), by simply cloning our [Github repository](https://github.com/vigarov/SmartGlass). The procedure of flashing is most easily done through ESP-IDF: the mandatory framework for anyone programming an ESP. You can find more information about it on [their website](https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/index.html): they mainly provide a step-by-step guide to getting it as a *VSCode* extension (more on that in the next section). Our repository is already setup for functioning with the extension flawlessly: once its installation is complete, simply *Build, Flash and Monitor* the code by pressing *CTRL+E* followed by *D*. Do not forget that depending on the model of the ESP you have, you might have to press on the *Boot* button to enable flashing. Furthermore, if an error similar to `*COM stream cannot be opened*` occurs, make sure you have selected the correct *COM* port your ESP has established connection to with your computer[[1]](#footnote-1).

Once the firmware has been flashed to the microcontroller, everything is set! You can simply enjoy the experience of the Smart Glasses by powering them, either through USB, or via a battery, if you have correctly followed our 3D design and hardware guide.

# Making your own firmware – Tips and Tricks

As you can see in our repository, our codebase is… **HUGE**. Do not let that frighten you! A very good result can be achieved by writing much less, although messier, code. Here is general advice I would give to anyone for an embedded system project:

* Start by testing each component you intend to use individually – for instance using an Arduino board, but any board works really. Doing this has several **major** advantages: firstly, it obliges you to get to know the components, and existing libraries around them. Using example sketches that can be found on the internet is great way to do so, and saves you much time in the long term of the project’s lifetime
* Scale only if needed. Indeed, this advice holds for any project, but for embedded systems programing in particular as well. Indeed, the more you modularize your code, to prepare it for scaling, the more it becomes complex, and your codebase will increase. Although preparing for scale can and is good for end-products or group projects[[2]](#footnote-2), it is not necessary for prototypes, or single person projects

Now let us get into the straight of the subject: what does our firmware implementation contain[[3]](#footnote-3)?

To answer this question, let us first take into account which constraints we faced, and which design choices resulted off them:

1. The main point is that the communication between the two principal actors: the Android application and firmware is to be as flawless as possible
2. The ESP32 has limited resources: it is considered as relatively low resource-consuming: any heavy computations are to be outsourced to the phone if possible

As such, to solve both, we must make sure to use *all* that is offered to us by the ESP: mainly both of its cores. This introduces further challenges as know we must bear into mind all parallelism and concurrency issue one could consider while writing such multi-tasked code. The design choice we have made is to have BLE handling completely separated: on CPU Core 0 (PRO\_CPU), whereas everything else would run off CPU Core 1 (APP\_CPU). Concerning the BLE, one can find a \*\*very\*\* handy recapitulation table in BLE/BLEHandler.h, summarizing all of the advertised

Now has come time to define what “everything else is”. Our firmware is portioned in the following way :

* a micrOS (uOS) handles all context switches and events that could lead to a modification in either the current application or what it is displaying, and forwards requests accordingly
  + Each application is stored by the uOS, and `resume`d() or `stop`ped() accordingly.
  + The application themselves perform computations, if needed, in a separate thread handled by the ApplicationContainer
* Displayable objects: this were the source of many, many, many hours of work on this project, as they are an abstraction to what a… displayable (on screen) object is. As such, every application is marked as a container of such objects. Some objects are mutable, others – not, this could simplify their implementations[[4]](#footnote-4). Having such abstractions, one can build complicated layers of graphical design by simply referencing other such displayables (which is what is done in most Containers, take a look at `Header` or `NavigationContainer` for instance), while maintaining easy change of state.
  + Furthermore, due to HyperDisplay’s library not supporting text printing on the ESP32, I also had to adapt a font so that we could display text: a crucial part of the project. This has further been done in a modular fashion to enable for any other font to be added quite easily, though we have not used that since adding different fonts is memory expensive (especially for bigger ones), and did not have a clear use case.

The details of all the implementations of the aforementioned structures and abstractions are left as a pleasure for the reader to discover by reading the code, once again, present in `victor\_idleApplication1`. Documentation of every single method and class has not been added, although crucial components have indeed been documented in the files themselves.

# Conclusion

Writing one’s own firmware is a challenging aspect of any embedded system project, but, in my opinion, one of the best parts of it! If you want to simply replicate the SmartGlasses as we had them, simply flash our firmware onto your microcontroller. Otherwise, feel free to learn from my mistakes to try and implement your own interpretation of the task at hand!

Good luck and happy hacking!

1. A tricky way to find that out is, on Windows, by opening `*Device Manager*` connecting, disconnecting, and reconnecting the ESP, and watching what values change in the Serial ports section [↑](#footnote-ref-1)
2. When doing group projects, communication is key: it is much more preferable to talk about implementation abstraction for 20ish minutes, than to jump into code that might end up useless and could’ve been avoided by having a better grasp of the big picture. Although this has, fortunately, not happened in our project, it is still to be kept in mind [↑](#footnote-ref-2)
3. As per the code used in the demo of 03/06, in the branch `victor\_idleApplication1` [↑](#footnote-ref-3)
4. I have ended up refactoring 3 times the concept of Displayables, each time approaching the different objects one to another (e.g.: now a ConstantContent is nearly the same as a Content), due to, mainly, **memory problems**: our ESP, after setting up all our tasks, including BLE, had only about 80kB free RAM, and trade-offs between computational efficiency, and memory efficiency had to be done, the latter being cause of many core panics complicated to debug, and whose only solution is…. A refactor. [↑](#footnote-ref-4)