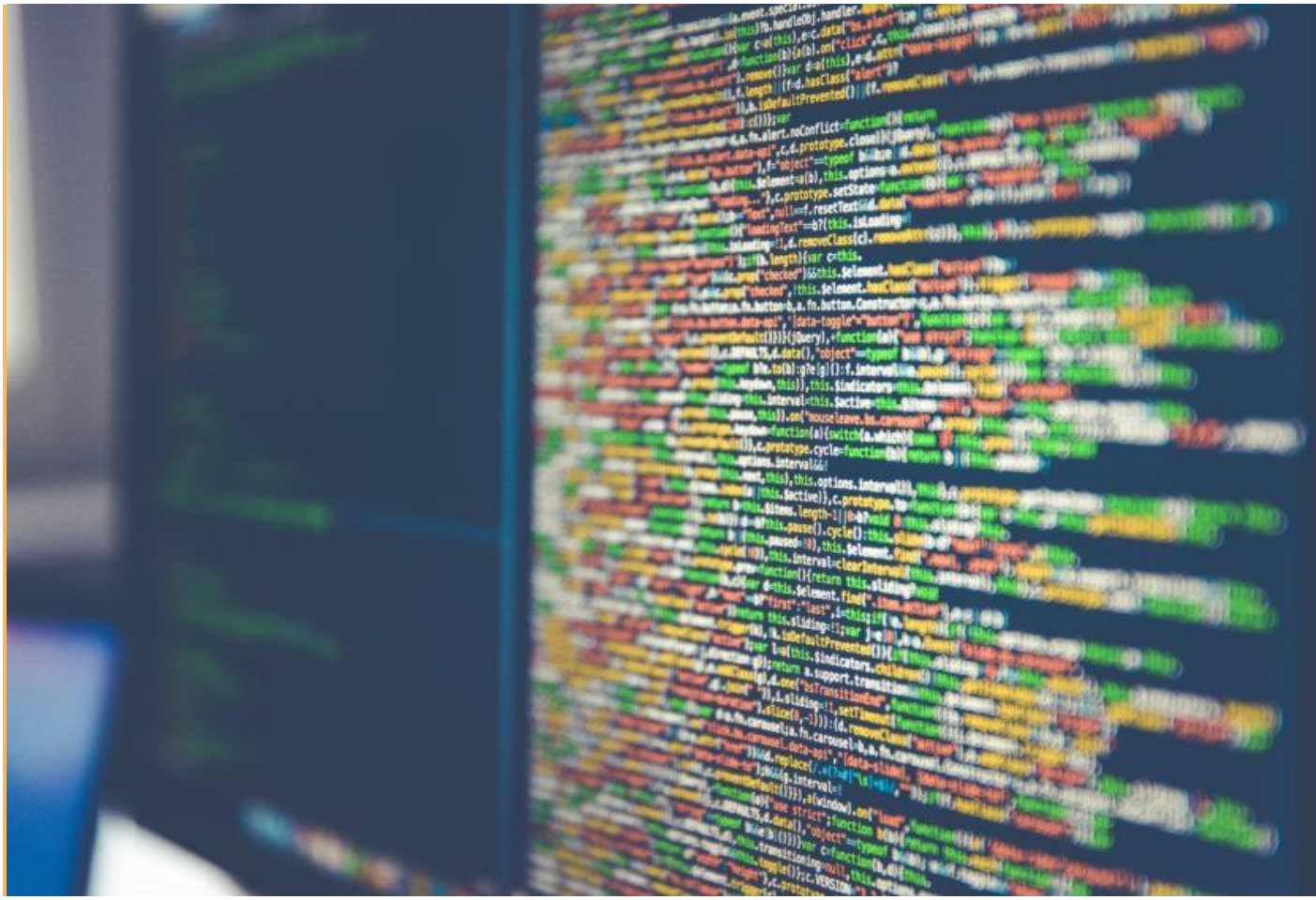
EMBEDDED **AMS**

Presents: Five habits for faster embedded software development.

By Bart Hertog, Embedded AMS

Five habits for faster embedded software development.

By Bart Hertog, Embedded AMS



Having been involved in several embedded software projects over the past years, I have picked up a number of misconceptions that can slow down development, and even been guilty of some of these myself!

In this article, I'll be highlighting the tell-tale signs of those common mistakes, and giving a bit of insight on how you can avoid them.

The list is in no particular order, but at the end of the article, I've shared a summarised list of good practices that will help you speed up your embedded software development processes.

1. It's only a small project, so there's no need to structure my code.

In my experience, small projects are always extended. We all know that they can quite easily get messy. To avoid this mess, separating hardware interfaces from your functional code can make a huge difference.

When you initialise hardware within a function or class that also holds functional code – like a state machine or calculation – something is bound to go wrong.

At some point, you'll either want to use the functional code or the hardware interface in some other part of your programme. In this instance, you'll wish you had separated the two, because now, you have to rewrite your code to separate the two.

"It is a really good practice to separate hardware interfaces from your functional code."

When you structure your code correctly from the start, you'll be left with simpler, extendable and easily maintainable code, which will ultimately save you time.

Why not spend just one hour a day refactoring? Cleaning up your code today will truly help you **speed up development** tomorrow.

2. We can't test the code, the hardware is not ready.

It's always a challenge to develop custom hardware and software simultaneously.

If software development has to wait for hardware to be completed, the development of the Internet-of-Things (IoT) products could take twice as long.

One of the best strategies for simultaneous development is to use development boards for software testing, and as a reference design for the hardware.

"When the new printed circuit board (PCB) has been produced, the software team should be able to use the software they have developed at set-up, with development and breakout boards WITHOUT modifications!"

Almost every range of chips has a development or breakout board, and they are often very cheap. When you begin the development cycle, try to follow these steps:

1. A **proof of concept is required at the start of a project,** both for your selected hardware and for the performance of the software. Use the available breakout and development boards to capture the first portion of your functional requirements. As little as 50% will get you a long way.

   Connect the PCB's using breadboard wires or flat cables. Now, to improve the reliability of this setup, mount it to a fixed board. The cable connection will be stable when the different PCB's can't move in relation to one another. Use whatever works best for your PCB: screws, double-sided tape or even glue.

2. Let the software team develop the code for the breakout test setup concurrently to the hardware team. This will enable them to capture the current set-up in a PCB design (the only design for the features currently developed by the software team). To make the PCB usable to test the next features, add some headers you can connect additional breakout boards to.

3. Once the new PCB has been assembled, the software team should be able to apply the software they have developed on setup with development and breakout boards, WITHOUT modifications! When everything still works, the PCB and software has passed the test.

4. You can start adding new functionality at this point, starting again at step one. Use the new PCB to build on, using the extension headers. When you've covered all the features and additional hardware needed, remove the extension header from the PCB and your final design is complete.

3. Not reusing excising code.

Something that is really common in other areas of software development is the use of libraries. But it's not that common in embedded software development. A possible cause for this is the wide variety of platforms and chip types that don't support the same specifications.

There is a great initiative that allows for easy reuse of software over different chips of ARM microcontrollers.

A few years ago, ARM itself started the MBED-OS platform. The platform is evolving, and has turned out to be extremely usable for professional products. MBED-OS is a software library which relieves the programmer from having to write chip specific code. By supporting a wide range of different ARM chips from different manufactures, it becomes possible to write generic code.

For example, in MBED-OS, the I2C bus has a generic interface. This allows for the *development of drivers for external chips, which are connected to the I2C bus.

The driver doesn't need to have any knowledge of how to setup the I2C bus. It can call the function "write()" to send data over the bus, just as it can use "read()" to receive data.

I recently used this to integrate a commonly used real-time clock (RTC) chip into a project. Since I didn't have to rewrite code for all the features of the chip, it was done in less than an hour. Now that's what I call **effective embedded development**!

4. I wrote this code myself. I know what I did, and don't need to document it.

Well that is true, but only right now or five minutes after you've completed the code. Give it a month or two and you'll have to spend time reading through the code to understand it again.

Now, I am not just talking about adding a line of code which states "This function calculates absolute speed using Pythagoras". I'm talking about documentation which gives proper context. Here is an example of limited documentation:

```
//! This function calculates absolute speed using Pythagoras.
float speed(const float x, const float y);
```

And now some more elaborate documentation:

```
//! This function calculates absolute speed using Pythagoras.
/*!
Use it to calculate the energy of the object. It is not suitetablee
to calculate the position.
\param [in] x The velocity of the object along the x axis in [m/s].
\param [in] y The velocity of the object along the y axis in [m/s].
\return The absolute velocity of the object is returned in [m/s].
*/
float speed(const float x, const float y);
```

Please note that I use Doxygen style documentation.

Don't just document the functions, but also the bigger picture on how to use a set of functions or class.

5. Writing your own serialisation code

Ever been in a situation where you just needed to exchange a handful of commands with a microcontroller, for it to send back only a small amount of data?

If this sounds familiar, you probably know what's coming next.

Yes! Format extensions! At some point, you'll want to extend the command or data structure. So, how do you keep your format synchronised with the rest of the team? It's not uncommon to need an update to your back-end or app at the same time, to keep things working.

There are many ways do this. You could parse raw structures and make sure you have the same binary structure on the receiving side.

Any change on either side will brake communication immediately. You could use text-based self-describing formats like XML or JSON, but they are not very memory-efficient, and have a small footprint in microcontrollers.

Our preferred means of exchanging data is to use a language-neutral, platform-neutral, extensible mechanism for serialising structured data. That's why we use the Protocol Buffers format by Google. With its wide range of supported programming languages, this makes it extremely easy to exchange data between totally different platforms. For embedded targets, we use the Nanopb, which features a small flash and memory footprint.

The way to use this is by defining the data you want to exchange in a separate file, a *.proto file:

```
enum State {
  Idle         = 0,
  Initializing = 1,
  Run          = 2,
  Stop         = 3
}
```

```
message Update {
  int32 count = 1;
  float speed = 2;
  State state = 3;
}
```

As you can see, an update message is defined, which communicates an integer float and an enum value.

This file is then processed by the protobuf compiler, which will generate source code for you. This is the bit that makes your life so much easier.

All you have to do is update the message definition in the proto file and get the code – without any work. I'd suggest you place the proto files for a project in a separate repository.

Each of the other repositories can then include this repo as a sub-module, allowing each team member to edit the definition file and push their changes to the rest without breaking any builds.

6. Summary

Let's summarise the practices you could follow to improve your embedded software development speed:

1. Structure your code from the start.
2. Develop your hardware and software iteratively, using prototyping PCB's.
3. Reuse software where you can.
4. Document the overview.
5. Use a language-neutral, platform-neutral, extensible mechanism for serialising structured data.

I trust this document will help you when developing your software for embedded, IoT and industrial applications. If you have any tips or remarks to add, please send us an email to info@EmbeddedAMS.nl.