Practical Guide to Bare Metal C++

Alex Robenko

Version 1.0

Table of Contents

Overview

Once in a while I encounter a question whether C++ is suitable for embedded development and bare metal development in particular. There are multiple articles of how C++ is superior to C, that everything you can do in C you can do in C++ with a lot of extras, and that it should be used even with bare metal development. However, I haven't found many practical guides or tutorials of how to use C++ superiority and boost development process compared to conventional approach of using "C" programming language. With this book I hope to explain and show examples of how to implement **soft** real time systems without prioritising interrupts and without any need for complex real time task scheduling. Hopefully it will help someone to get started with using C++ in embedded bare metal development.

Audience

The primary intended audience of this document is professional C++ developers who want to understand bare metal development a little bit better, get to know how to use their favourite programming language in an embedded environment, and probably bring their C++ skills to an "expert" level. Why **professional**? Because bare metal platform has lots of limitations. In most cases no exceptions and no runtime type information (RTTI) support will be available. In many cases the dynamic memory allocation will also be excluded. In order to be able to use C++ effectively you will have to have deep knowledge of existing C++ idioms, constructs and STL contents. You must know how your favourite data structures are implemented and whether it is possible to reuse them in your environment. If it is not possible to use the STL (or any other library) code "as is", you will have to implement a reduced version of it, and it is better to know how the library developers implemented the feature and how to make it work with the constrains of your environment.

The professional embedded developers with intermediate knowledge of C++ may also find this document useful. They will probably benefit from lots of C++ insights and will have several "eureka" moments with "I didn't know I could do that!!!" kind of thoughts.

If your C++ knowledge doesn't go much beyond polymorphism and virtual functions, if template meta-programming doesn't mean anything to you, probably you are not ready to use C++ in the embedded environment and this document will probably be too complex to understand.

I'd like to emphasise the fact that this is NOT a C++ tutorial. There are lots of resources on the web that teach conventional C++ with OS services, exceptions and RTTI. My personal opinion is that you have to master C++ in regular environment before using it effectively in the bare metal world.

C++ Popularity

C++ is quite popular in the embedded world of Linux-based embedded systems. However, it is not that popular in bare metal development. Why? Probably because of its complexity. Knowing C++ syntax is not enough. To use it effectively the developer must know what Standard Template Library (STL) provides, what can and what cannot be used when developing for specific platform. STL mastery is also not enough, the developer should have some level of proficiency in template meta-programming. Although there is an opinion that templates are dangerous because of executable code bloating, I think that templates are developer's friends, but the one must know the dangers and know how to use templates effectively. But again, it requires time and effort to get to know how to do it right.

Another reason why C++ is not used in bare metal development is that software in significant number (if not majority) of projects gets written by hardware developers, at least in its first stages just to make sure the hardware works as expected. The "C" programming language is a natural choice for them. And of course majority of hardware developers lack proficiency in software development. They may have some difficulties writing code of good quality in "C", not to mention "C++". After software reaches certain level of complexity it is handed over to software engineers who are not allowed to re-implement it from scratch. They are told something like: "This code almost works, just fix a couple of bugs, implement this short set of features and we're good to go. Throwing away the existing code is a waste, we do not have time to re-implement it."

The last reason, I think, is psychological one. People prefer to be wrong in a group than right by themselves. When majority of bare metal products being developed using "C", it feels risky and unnatural to choose "C++", even though the latter is better choice from the technological perspective.

Benefits of C++

The primary reason to prefer C++ over C is **code reuse**. Thanks to templates, it is much easier to implement generic piece of code that can be reused between projects in C++ than in C. When implementing everything from scratch, then probably using C++ instead of C won't give any significant advantage in terms of development effort, maybe even extend it. However, once generic components have been developed, the whole development process for next projects will be much easier and faster, thanks to reuse of the former.

Contents of This Book

This document introduces several concepts that can be used in bare-metal development as well as shows how they can be implemented using features of latest (at the time of writing) C++11 standard.

The code of generic components is implemented as part of "Embedded C++ Library" project called "embxx" and can be found at https://github.com/arobenko/embxx. It has GPLv3 licence.

There is also a project that implements multiple simple bare metal applications using embxx which can run on RaspberryPi platform. The source code can be found at https://github.com/arobenko/embxx_on_rpi. It also has GPLv3 licence.

Both projects require gcc version 4.7 or higher, because of C++11 support requirement. They also use CMake as their build system. The code has been tested with following free toolchains:

- GNU Tools for ARM Embedded Processors on Launchpad
- Sourcery CodeBench Lite Edition

The whole document is ARM platform centric. At this moment I do not try to cover anything else.

To compile Raspberry Pi example applications in Linux environment use the following steps:

- Checkout embxx_on_rpi project

```
> git clone https://github.com/arobenko/embxx_on_rpi.git
> cd embxx_on_rpi
```

- Create separate build directory and cd to it

```
> mkdir build
> cd build
```

- Generate makefiles

```
> cmake ..
```

Note that last parameter to cmake is relative or absolute path to the root of the source tree. Also note that embxx library will be checked out as external git submodule during this process.

- Build the applications

```
> make
```

- Take the generated image from `<build_dir>/image/<app_name>/kernel.img`

The CMake provides the following build types, which I believe are self-explanatory:

- None (default)
- Debug
- Release
- MinSizeRel
- RelWithDebInfo

To specify the required build type use `-DCMAKE_BUILD_TYPE=<value>` option of cmake utility:

```
> cmake -DCMAKE_BUILD_TYPE=Release ..
```

If no build type is specified, the default one is **None**, which is similar to **Debug**, but without `-g`