Refactoring existing MBARI Coastal Profiling Float (CPF) C# code to C/C++ Statement of Work (SOW)

V0.0	2020 Dec 17	Initial draft
V0.1	2020 Dec 22	Edited design guidelines section
V0.2	2021 Jan 19	General edits
V1.0	2021 Feb 06	Initial release
V2.0	2021 Jun 05	Changed target processor, added initial phase to SOW, reformatted SOW
V3.0	2021 Dec 08	De-scoped work to include just design review and building a simple foundation project
V4.0	2022 Jul 13	Updated to reflect progress integrating UARTs
V5.0	2022Nov05	Added phase 2

1. Introduction

MBARI has developed an autonomous oceanographic coastal profiling float (more at https://www.mbari.org/coastal-profiling-float/) that has been working relatively well for the last several years. The existing software is written in C# using the .NET Micro Framework and the Visual Studio IDE. In order to move this technology to the greater oceanographic research community outside of MBARI and for a variety of other reasons, we are going to rewrite the code in C/C++. MBARI works with a very small engineering staff and engaging an outside, qualified consultant(s) for this of work makes great sense. The goal for this work is for the consultant to provide a well-engineered VisualGDB (VGDB) "foundation" solution including a test framework and static analysis tools focusing on the basic framework and peripheral drivers. The idea is for the foundation solution to provide an architecture and most of the classes need to implement the full CPF functionality. Adding the additional code to provide all the required CPF functionality should be mostly a matter of extending the existing functionality without have the re-write very little (or none) of the delivered code.

The current plan for this work consists of 2 steps. The first step is a review of the work done to date refactoring the existing C# software into C/C++. The second step is turning the current code into well-engineered VGDB solution including a test framework and static analysis tools. Some introductory information is provided below. Work packages for the first 2 steps along with deliverable are described in some detail. If these 2 steps go well, as we expect, there are some examples of additional work packages that we may wish to pursue in the future.

It is worth stating up front that while we have a working Visual Studio C# solution and experience with C/C++ we are open to the consultant's suggestions in any area where improvements can be made. Furthermore, everything in the statement of work is up for discussion.

2. Background

2.1. Hardware

Currently, the target processor for the C/C++ app is a STM32H7A3. There are a variety of reasons why I've chosen this MCU and it would be worth reviewing this decision during the design review. The existing C# app runs on a GHI G400S SOM (https://docs.ghielectronics.com/hardware/netmf/g400s.html) with an Atmel SAM9X35 running at 400 MHz (which is probably much more processor than we really need). The G400S does have 64+ MB of SRAM and 4MB of flash and we take advantage of a significant amount of this. We are in the

process of developing a custom STM32H7A3 based SOM with external SRAM, flash and a uSD card that will eventually be the target hardware. While this SOM is being developed, we have been using STM32H7A3 Nucleo boards and/or STM32H7B3 Discovery boards as needed.

For information, the CPF currently has 6X UART connected devices, 2X spare UART ports, 3X core I2C devices and 9X secondary I2C devices. Most of the UART devices are connected to native UARTs on the MCU but the existing hardware also includes additional I2C UARTs. All 12X I2C devices are multiplexed onto one microcontroller I2C port. There is also a uSD card on a 4 bit wide SDIO port. The new code should take advantage of the additional native UARTs, I2C and SPI ports available on the STM32H7A3.

2.2. Software

The current C# app uses a non-blocking state machine with minimal interrupt service routines for asynchronous device communications. There are 3 basic types of classes: states, devices and peripherals. The basic states for the CPF are SMNotRunning, initializePlatform, initializeMission, preMissionDelay, recovery, initProfile, findNeutral, descend, park, anchor, startCPMode, deanchor, ascend, stopCPMode, surfaceOps and quit. Each state inherits basic functionality from StateBase. Each device is basically an external piece of hardware e.g. Iridium modem, motor controller, microSD card or scientific instrument, that interfaces to the CPF app via a standard microcontroller peripheral e.g. UART, I2C and possibly SPI in the future. As an example, the most important scientific instrument is a Seabird 41 CTD (conductivity, depth, temperature). In the C# code, there is a singleton CTD class that provides the functionality required to control the CTD and this class inherits a UART class.

3. Software Engineering guidelines

The highest priority goal is to provide robust, well tested code that can run on an autonomous platform for years at a time. The second highest priority goal is to provide code that is easily understandable and maintainable by MBARI engineers upon delivery and also years down the road. We are very pragmatic in our approach to software engineering and development and are happy to do whatever makes sense but we've found that appropriate design guidelines can help. The following are just examples of the kind of design guidelines we'd like to formalize.

3.1. Code should not require an expert understanding of C/C++.
3.2. Classes, methods and variable should use very descriptive names. Use verbs in method names to describe the action being performed e.g., checkForStuckPressure, logEngrMsg, isQEmpty.
3.3. RAII should be used where ever it makes sense.
3.4. Dynamic memory allocation e.g., new, should only be used during initialization
3.5. My personal order of preference is std::array, string, std::vector, c-style arrays
3.6. Encapsulation is important. For example, peripheral drivers shouldn't care what device they're connected to and visa-versa.

4. Work Packages (WPs)

WP 1. Review of existing code

I've been developing a small C/C++ prototype of our existing C# code with the hope of improving my knowledge of C/C++ and prototyping the critical parts of the existing code. I've used a STM32H7A3 Nucleo board and a STM32H7B3 Discovery board for this work. This first

WP would review this prototype with the intent of 1) introducing the consultant to the existing software design and 2) identify ways to improve the design.

The current prototype code implements the state machine with a few states, a couple of UART device interfaces, FATfs on a uSD card and some real time clock functionality. The code works but there's a lot of cut corners basically because I don't know enough about properly structuring an embedded C/C++ program. The prototype code is somewhat object oriented but is littered with //TODO comments that identify areas I know aren't right.

Deliverables:

a. List of changes and/or additions to MBARI prototype. If an entirely new approach is needed, that's fine too. This can be a combination of comments in the code, a document with more extensive explanations, architecture diagrams, etc.
b. An approach for unit testing and some examples of the kind of unit tests we should be developing.
c. Suggestions for how to integrate static analysis and potentially an example of running an appropriate static analysis tool on the existing C/C++ code.

WP 2. Initial project improvement

The goal of this WP is to divide up the list of changes identified in WP1 between EA and MBARI and start implementing the changes and additions consistent with the budget proposed by EA. The idea is this updated prototype code will serve as the foundation for our future work with the hope that most of the required classes and hardware interfaces are in place and just need to be extended to cover additional IO and the additional states required to operate the CPY.

Deliverables:

a. An embedded C/C++ VGDB project running on a STM Nucleo H7A3ZI-Q with the same functionality as the exsiting code but with the changes and additions from the list in WP1 implemented. More specifically
   i. Some idea about how to address the issues identified by the TODO comments. In particular, I'm pretty sure I'm using static variables incorrectly.
   ii. I've hacked together some STM32 HAL drivers for the hardware interfaces in a couple of "classes" but I'm assuming this can be done much better than my hack.
b. A test framework like CPPuTest, with several example tests
c. An example of what might make sense for static analysis.

The following are examples of work packages we may want to consider in the future.

WP 3. I2C Drivers

This is basically the same task as WP3 for the I2C peripherals. We can provide the details of the I2C devices at the beginning of the work.

WP 4. External RAM

WP 5. External Flash

WP 6. Watchdog

Phase 2

WP 1. Modify the existing state machine to be event driven

The goal is to have an event driven state machine that handles the current list of events and when the list is empty, puts the system in an appropriate low power mode or the existing system::delay(N) mode. Here are some thoughts on implementation details:

1. An event, like a character arriving on any UART or maybe a complete LF message arriving on a UART should wake the system from sleep.
2. We've talked about an event queue to hold the list and that still makes sense to me.
3. We might want a function that sets the specific sleep mode from the available H7A3 sleep modes.
4. We probably need to use the LPTimer or one of the watchdogs or something similar to wake after a specified period no matter what.
5. For sleep modes that don't maintain the memory, we need to store some state information in the controller battery backed SRAM. As a start, profileNum is one such state variable.
6. Currently, the C# code has the ability to turn off the power to everything but an external low power timer. The external timer turns the power back on after a user defined period. When the period expires, the timer powers up the system, the system checks the time and if it is time to do the next thing, like start the next profile, it does that. If it isn't time, it turns the power off again. I'm thinking that his functionality can be rolled into the event driven architecture. That is, we can go into the lowest power state and the RTC (which is in the battery backed power regime) wakes the system up after X minutes.

WP 2. Add an I2C Driver

Very much like the current UART driver and level of abstraction. If DMA seems useful, that would be great. We will provide 2 devices as examples: 1) an energy monitor development board. This device is a pretty normal I2C device, if you want a piece of data from a specific register you do that in one transaction. 2) The current GPS receiver has an I2C interface in addition to the UART interface. This gizmo has a slightly weird I2C interface. You read from the same register over and over again until you get a complete message. The first read from that address tells you how many bytes are in the complete message. The GPS receiver is the only device that we use now or any device I can think of that we may use in the future that has a choice of interface. So, whether to add another layer of abstraction to the GPS class that allows for a UART or I2C interface is up for discussion.

WP 3. Review MBARI generated code

My plan right now is to have a second CPF team member be developing code in parallel with your efforts. We should plan for an initial chat to discuss general guidelines and one or 2 follow up sessions to review progress and adjust path as necessary. This should actually be a really good way to figure out how to effectively document what we're doing so MBARI personnel can continue developing the new code. The tasks I'm thinking of right now include the following.

1. Add a buoyancy engine class with the code to handle the control loops and the other functionality associated with the buoyancy engine
2. Add all the functionality in the current C# initProfile and initMission classes to the C++ code

3. Once the I2C driver is complete, finish the EnergyMonitor device class with the existing C# functionality
4. Eliminate the _discovery files. Given that PJ has started this process, it may be more effective for him to continue?

WP 4. Training session

Please reserve 3 hours (or so) at the end of your Phase 2 work for a training session for the CPF team. The goal is to review and explain the entire new code base so the CPF team can continue developing code with the same architecture, level of abstraction, style, documentation, etc. As of now, the CPF team is Gene and 2 other engineers.