# Hardware manager's lessons learned

Nils Brynedal Ignell
nbl08001@student.mdh.se

## ABSTRACT

In the beginning of the Naiad AUV Project I was assigned to be the manager of hardware development during the project. In this report I have written down my experiences and what I learned. My intended readers are primarily future students of this project course but also anybody else who participates in projects, primarily student projects, that include many people and span over longer periods of time.

## 1. INTRODUCTION

In the beginning of the project, our supervisor decided that the project group was to be divided into a software group and a hardware group. Our supervisor also assigned four roles of leadership as follows: One overall project manager, one manager for the hardware group (me), a manager for the software group and one person responsible for documentation.
The person assigned as software manager was also assigned to be responsible for sponsorship, which in practice came to mean that he devoted the vast majority of his time to sponsorship and consequently there was no software manager. I will come back to this later.
The role of the person responsible for documentation was pretty much abolished as well and the responsibility of documentation was spread over the whole group. This did however not give any apparent negative effects so I will not go into more detail about this.

One of the first things the four managers did was that we interviewed each of the other members of the group to learn who they were, what they were good at and if they wanted to be put in the software or hardware group. These interviews gave a good overview of who the individuals in the group were and which role should be given to whom. *I strongly recommend the managers of the next year's project to do the same.*

One open question that was not decided by the supervisor was whether *firmware* was to be considered software (and consequently the responsibility of the software group) or hardware (the responsibility of the hardware group). This question, and the question where to draw the line between firmware and software, was to give significant problems to the project.
One obvious way would be to say that any programming (software or firmware) would be done by the software group. The problem with this way would have been that many parts of the robot include both hardware and software. For example, the INS controller (that handles the inertial measurement unit and fiber optic gyroscope) consists of both hardware (the sensors themselves and interface circuits) and also firmware for reading the sensors, sending CAN messages and so forth. In light of this fact, putting the hardware development and firmware development in two different groups did not look like the best of ideas.

Firmware was decided to be under the responsibility under the hardware group. As for the division between software and firmware, the Mission control, Motion control, Simulator, the vision system was considered to be software and more or less any other form of programming was considered firmware.
I did initially support this, mainly since I foresaw that there would be many times when hardware and firmware needed to be developed together and that there would often be significant interdependence between hardware and firmware. Also, I believe that some part of my ego also wanted me to have the responsibility (and therefore reasons to take credit) for work that also contained some programming.
However, as the project went on, I came to regret this and I realized that the division of work should have been done in a different way.

## 2. LESSONS LEARNED
Here I have listed the lessons I learned during the project that I want to share with others.

### 2.1 Leave your ego at home
Throughout the course of a project like this you will participate in many discussions. Many times there will be people that think differently than you do. More importantly, *there will be times when you are wrong.* When those who oppose you have valid arguments, you have to reconsider if you really are right.
Don't be afraid to acknowledge that you are wrong. Keep an open mind and focus on factual arguments rather than taking anything personal.
Remember that design choices should never be determined by the egos of individuals in the group.

### 2.2 Listen to your colleges
Early on I realized that even though I was the manager of the hardware team, I still did not know particularly much more than anybody else in the team. Therefore I frequently discussed the decisions I took with my colleagues and I was very clear that I wanted feedback. If somebody realized that I had taken a bad decision then I wanted that person to tell me this so that I could make better decisions, etc.
I believe that this was a very good thing to do. There were many times when members of my team come to me and told me that a decision I had made were not the best, which often helped to avoid many mistakes.
This lesson is closely related with Section 2.1.

### 2.3 Define roles of management and follow them
As mentioned, there was in practise no clear manager over the software group. This does not, to me, seem to have affected the work of the software group itself in any negative way. However this had in fact an effect on the project as a whole.
During the project, I often needed to talk to the software group about how the firmware should interface with the software. An example of this is the Motion controller (software) that uses Motor controllers (firmware and hardware) to control the motors, I saw this as the Motor controllers offering a service to the Motion controller.
Whenever I needed to talk to the software group about a certain subject I would to talk to the individual that was responsible for this particular subject, which meant that the person I talked to didn't have an overall understanding of the software group's work and I never really got the the whole picture. Had there been a software manager, I could have spoken to him/her and we would together have had a much better over-all picture of how the firmware-to-software interface should be.

The interface between hardware and software should be discussed between the hardware and software managers (and also possibly the overall project manager) and not between individuals in each group. Please see Section 2.7 for more about this.

### 2.4 Define software standards and follow them
Regarding software standards, I am not too concerned about coding styles such as whether one should use underlines or not (e.g. *my_variable* or *myVariable*), even though these types of standards do improve readability.
More concern should be put into what the overall structure of the whole body of code should look like. This includes a folder structure (e.g. one folder for hardware specific drivers, with subfolders for each hardware, one folder for platform independent code, and so on) how different parts of the code depend on each other and so forth.

Please note that these definitions will in turn be further defined in Section 2.6 and that this lesson also is associated with Section 2.5.

Once the software structure and coding standards are decided, one has to make sure that they are followed properly. I advice the Software manager to read all code that his subordinates write. This will enable him/her to ensure that the code is written in a good way. This will also give the Software manager a better overview of all the software that is written during the project.
If somebody thinks there should be changes made in the coding standards or the structure of the software, then this should be discussed in the whole project group and a decision should be made. Individuals should avoid to deviate from the decided standards at their own initiative.

### 2.5 Use a top down design flow
A system such as an AUV, or any other robot developed during this course will be a system of subsystems where the subsystems are dependent of each other. For this reason it is important to first define the subsystems and their interfaces before starting to develop each subsystem.
If one does the opposite, first developing a subsystem and then define its interface, then this subsystem will have to be finished before any other subsystem (that is to interface with the subsystem) can be developed.

### 2.6 Clearly define each subsystem and its interface
As mentioned in Section 2.5, in a project of this size there will be many different subsystem that will be created by different people but still have to work together in the end. An example of this is the Motion Control subsystem that interfaces with each Motor controller that in turn controls the motors.
It is of the utmost importance that the person(s) that build one subsystem (i.e. the Motion Control) knows what to expect from the subsystem(s) (i.e. the Motor Control) that their subsystem uses. There has been several instances throughout the project where one subsystem has been created to solve one function and those who are to use this subsystem have understood that it would solve a slightly different task.

I recommend defining a set of tests for each subsystem at an early state, before the subsystems are developed. This will both help the person(s) developing the subsystem and help others understand what the subsystem will do. Defining User stories [6] and/or Integration tests [4] that will be used later on in the development cycle (testing phase) are good ways for the project team to define the goals for the different subsystems.
I also encourage that the process of defining each subsystem,

their interfaces and tests is done by several people and that what is decided is then presented to the whole project group so that anyone can come with suggestions or ideas.

## 2.7 Firmware is software

As mentioned, there were several possible definitions of firmware and firmware could be considered either software or hardware. In any case, each piece of firmware will need to interface with to two things: the hardware (the Sensor controller needs to interface to its sensors, the Motorcontroller needs to interface with the motors, etc.) and to the software (e.g. Motion control, Mission control, Simulator, etc).
I have come to realize that the firmware-software interface is more complex than the firmware-hardware interface. The reason for this is that the firmware-hardware interface is specific to each subsystem, for example: the person writing the Sensor controller firmware only needs to talk to the person building the Sensor controller hardware (choosing sensors and building interface circuits, etc). Whereas the firmware-software interface is more general. This is discussed in Sections 2.4, 2.5 and 2.6.

## 2.8 Have a person responsible for sponsorship only

The work of getting sponsors is enough work for at least one person throughout the project. This person should not have other duties.

## 2.9 Individuals will make mistakes

The well known fact that ”To err is human” has been apparent during this project, not only others but also myself have made mistakes that have had negative consequences for the project.
I have realized that when working in a project such as this project course, one has to remember that any individual can make mistakes. For this reason one has to work in a way such that the mistakes of an individual do not become a mistake of the whole project group, mistakes need to be found and corrected. This can sometimes be hard to do in practice since assuming that a person could make a mistake can be interpreted by that person that one does not trust him/her. For this reason, this lesson is closely related to Section 2.1, each person has to realize that he/she can make mistakes and allow others to check his/her work.
For managers (software, hardware and overall managers) this lesson also closely relates to Section 2.2.

## 2.10 Talk to the client about requirements

During the project it became apparent that several of the ”hard” requirements given by the client were not so hard at all. Neither did all the requirements seem as thought through as they first appeared. Several of the requirements turned out to be more of ”nice to have” rather than ”need to have”, for example the robot-to-robot communication, IP-over-CAN, the use of space-plug-and-play, etc.

The same is likely to be true in following project courses. Consequently, I strongly advice future students of this project course to look carefully at the given requirements, analyze them so that their purposes are fully understood and prioritize them. Once this is done, requirements that either seem to be to hard to fulfil or that have low priority should be left for later evaluation. It is better to start with a few important requirements than trying to meet all from the beginning.
All this should be done in collaboration with the customer.

This lesson is closely related to Section 2.11.

## 2.11 Time is precious

There is a lot of work that needs to be done during a project such as this one. Even though each student is meant to put in a lot of time into the project, one will soon find that one will only have time for a fraction of what one first thought that one would be able to do.
One principle that was mentioned during the project was ”Take whatever time you think something is going to take, multiply by $\pi$ and round up”. This was said as a joke but in some instances it was not too far from reality.
For these reasons, one has to be very selective on what to spend time on. This applies for each person on all levels of the project.

## 2.12 Acquire equipment early

During the course of a project such as the Naiad AUV project several different kinds of equipment will be needed, some of which are crucial to the project's success. One such piece of equipment is a programmer for the microcontrollers used in the project. The Naiad AUV project used AT90CAN128 microcontrollers programmed by a JTAGICE MKII programmer.
Students of future projects should make sure to get access to programmers for the microcontrollers that are to be used as soon as possible. The Naiad AUV project had only access to one programmer, which proved to be far from enough.

As mentioned earlier in this report, testing is important. There will undoubtedly be times when things don't work. Unlike software development, where much testing can be done ”in software” using Unit testing [5] or simple text outputs to the screen, development of electronics and embedded software (or firmware, see Section 2.7) often requires hardware for testing.

Embedded systems are generally not connected to a computer screen. For this reason, one needs some other method to retrieve debug output from the system.
The simplest way for microcontrollers such as the AT90CAN128 used in the Naiad AUV to communicate with the ”outside world” (usually a human looking at a computer screen) is via UART. For this reason it is of vital importance to have access to some form of UART-to-USB converters, i.e. a way of creating a comport on the PC that is connected to the UART bus on the microcontroller. Terminal software such as Bray's Terminal [1] or CuteCom [2] will be needed for this.
Section 2.13 goes into more detail about this.

For debugging the CAN bus, which future student projects at Mälardalen University are also likely to use, hardware such as the Kvaser Leaf CAN-to-USB interface [3] will be needed to monitor subsystems on the CAN bus. This type of hardware can also be used to emulate one subsystem when testing another subsystem. For example, during this project we sent faked motor control messages to the Motor controller and checked that the motor's speed changed according to the control messages.

For testing electronics, equipment such as oscilloscopes, multimeters, function generators and voltage sources is necessary.

When a student project begins, make sure that you get access to the equipment that you will need as soon as possible. Make sure to remind those you request equipment (or other things) from very often, if not daily.
There will often be a delay from that you request to get something till you actually get it. Keep in mind that this delay is highly likely to be longer than you expect. When something gets delayed, make sure that you find out exactly what (or who) is causing the delay. This will make it easier to figure out how to speed things up.
Ask nicely, but make sure that you are firm and determined to get the equipment you need, be clear that you are expected to achieve very high goals and that you cannot do so if you don't have the resources needed to do so.

### 2.13 Think of testing when designing hardware

Make sure that you keep debugging and testing in mind from the beginning in the design process of any form of electronics. As mentioned in Section 2.12, testing and debugging embedded systems is often hard.

Remember to build the hardware to enable access to at least one UART bus on the microcontroller(s). This way you can output debug information to a PC (using a UART-to-USB converter, see Section 2.12) when the system is running.
When building circuits that will interface each other[1] it is a very good idea to connect their interface pins (the Tx and Rx pins in the case of UART) to some form of connector (simple break of header pins for example). This way you can monitor the communication.
In the case of UART you will connect the Rx pin (the pin that "listens") of your UART-to-USB converter to one of the pins of the UART bus. If you want to monitor both pins at the same time you will need two UART-to-USB converters.

The above will apply for prototype versions of a circuit board, but it might be a good idea to have this on the final version as well. For example, if someone in the future develops new software/firmware for your hardware they might very well have great use of being able to output debug information on a UART bus.

## 3. CONCLUSION

Project courses such as this project course are good since they let each student use his/her skills in a realistic context, cooperate with others and do something that has not been done before. Because of this it is important that each student has a solid base of skills in his/her area(s) before starting such a project and that cooperation within the project group works well.

[1]Such as the Inertial Measurement Unit that communicates over UART with the AT90CAN128 on the INS controller.

As for my advice for a plan for future projects I recommend to use the first two weeks for analyzing the requirements given by the customer and for State-of-the-Art research. Followed by two weeks for defining the structure of the robot, its subsystems and the interfaces between them. Throughout this process one should keep a close contact with the customer (the person(s) or entity that orders the robot) in order to make sure that the structure decided upon is what the customer wants.
Once all subsystems are defined, at least one person should be assigned as responsible for each subsystem (one person can be responsible for several subsystems).

The next phase of the project would then be to define the structure of each subsystem in order to get an initial understanding for how each subsystem should be implemented. When this has been done it might very well become apparent that one or several subsystems can not be implemented in the intended way and/or that the interface of a subsystem has to be changed. *If this happens, then this would be something good.* The fault in the overall structure of the robot can then be detected at an early state before any major implementations have been done and consequently lots of time can be saved this way.
After this phase comes the actual implementation phase. Testing should be done continuously so that any errors or flaws are detected as early as possible. When several subsystems that are to work together have been implemented, one should try to test them together to verify that their common interface works (that they work together). As more and more subsystems are finished, more and more complex (and accurate) tests can be done.

Lastly, remember to document all work in a proper way. Your work is of little use if nobody else can understand or use what you have built during the project.

## 4. REFERENCES

[1] Bray's terminal download page. `http://hw-server.com/terminal-terminal-emulation-program-rs-232`. Accessed 2014-01-11.

[2] Cutecom ubuntu documentation. `https://help.ubuntu.com/community/Cutecom`. Accessed 2014-01-11.

[3] Kvaser can-to-usb products. `http://www.kvaser.com/en/products/can/usb.html`. Accessed 2014-01-11.

[4] Wikipedia: Intergration testing. `http://en.wikipedia.org/wiki/Integration_test`. Accessed 2014-01-08.

[5] Wikipedia: Unit testing. `http://en.wikipedia.org/wiki/Unit_testing`. Accessed 2014-01-08.

[6] Wikipedia: User story. `http://en.wikipedia.org/wiki/User_story`. Accessed 2014-01-08.