Software Engineering in the Mariokart System

Wim Looman
wgl18@uclive.ac.nz

Coauthors: Simon Richards, Zachary Taylor and Henry Jenkins
scr52@uclive.ac.nz zjt14@uclive.ac.nz hvj10@uclive.ac.nz

Supervisor: Dr. Andrew Bainbridge-Smith
andrew.bainbridge-smith@canterbury.ac.nz

Department of Electrical and Computer Engineering
University of Canterbury
Christchurch, New Zealand

***Abstract*—Something amazing about engineering a software system.**

I. Introduction

A. Software Engineering

Since this report is aimed at an engineering audience most of you will believe that a description of Software Engineering is not really required. Unfortunately true Software Engineering is relatively unknown, especially in programming courses run in Electrical departments around the world. That is not to say that Computer Science departments do a better job of teaching it, in fact Software Engineering really should be taught as a subset of Engineering [1], just that the style of programming taught to Electrical students is generally light on following the engineering practices that the rest of their courses rely on.

So, what is Software Engineering? It is simply the application of standard Engineering practice to the development of software. However because of the nature of software as a much more fluid abstract thing than the normal circuits designed by Electrical Engineers the precise method of application has to be changed.

At the same time as being more abstract than a circuit software is also much more concrete; there are no (or at least very few) annoying real world effects directly on the software. Assuming the circuit a microprocessor is in has been designed well the Software Engineer can take it for granted that the digital I/O used by something like a Inter-Integrated Circuit (I^2C) is basically a perfect connection straight to the internals of another device. Internally if there are no weird defects in the microcontroller you can assume that a function like:

```
int return_three() {
    int three = 3;
    return three;
}
```

will always return exactly 3. Not 2 when the batteries start running low, not 4 when it is a particularly hot day, always exactly 3.

This exactness of software enables the use of a few techniques that are not normally available in most engineering professions. For example it is possible to perform exhaustive testing and/or modelling of the system within acceptable time.

The major components of software engineering that will be discussed in this report are: version control, unit testing and continuous integration. Version control is probably the aspect of software engineering that is best applied by current engineers, however most still use an old system such as Subversion despite their being much better alternatives like Git and Mercurial available. Unit testing is a developmental practice that has been seeing a major increase in use for traditional software in recent years. This is largely because of development processes such as Test-Driven and Behaviour-Driven Development evangelised by the Agile Software Development proponents. Continuous integration is a major aspect of these newer software development methods where quality control is continuously applied to the system while under development, normally utilising unit testing as the main quality assurance system.

A lot of this is standard practice in software development shops. Unfortunately despite the large amount of code written by other engineering disciplines the same level of engineering practice they apply in designing their circuit board, concrete floor or ethanol extractor doesn't get applied to the code they develop in pursuit of these goals. This is most relevant to embedded development where the entire range of software engineering practices can be applied; some take a bit more effort because of the lower abstraction level, but they are all applicable in some way. Parts of this are also relevant for the other engineering disciplines, Matlab may not be a real programming language, but when developing simulations in it proper software engineering practices should still be followed.

B. Mariokart

The system on which this report will base most of the examples was codenamed Mariokart. This was a final year project for the University of Canterbury's Bachelor of Engi-

neering degree carried out by the authors. The overall goal of the project was to take one of the electric go-karts the department had and retrofit a drive-by-wire system on to it with an overall goal of having the kart autonomously drive around the campus. For the purposes of this report the main details of the system developed are:

- The overall design is a distributed system with 5 boards:
  - One for communication with a host laptop.
  - One for steering.
  - One for brakes.
  - One to interface to the motor controller.
  - One for collecting data from a variety of sensors.
- Each board is running an Atmel SAM7XC microprocessor.
- Communication between boards is carried over a Controller Area Network (CAN) bus.

For more details see *Embedded Hardware Design For Autonomous Electric Vehicle* by Henry Jenkins [2].

II. Background

A. Version Control

Out of all Software Engineering practices Version Control is definitely the most widely used by normal engineers. Unfortunately it is still not used everywhere. For example, you have just finished looking at a new power regulation system that you want to switch to for your next PCB revision; now you need to write a quick report on why it is so much better than the current system that you should spend all this time changing your design. What's the first thing you should do, open a new Microsoft Word document? Load up your LaTeX editor? *No*, you should initialise a new repository or ensure you have the projects documentation repository available and updated. Anything and everything that is more than a few lines long, or will ever be shared with a team member should be under version control.

This is very important for a multitude of reasons. Firstly it provides you with a time line of development activity. If you need to revisit a decision months later you can identify exactly when the initial review was made and when any revisions happened.

Secondly it provides you with a safety net. The following anecdote is a very good example of why this safety net is important.

> "A younger programmer asked an elder about his code and his coding style, and how the older programmer would do certain things. The older programmer said 'Let's take a look at your code', so the younger took out his laptop, opened his editor, and showed him.
>
> The older programmer looked at the code, thought about it for a bit, and then started editing it. He deleted the class internals, leaving only the structure, and then rearranged the structure, saying 'Here's how I would do it to make it more efficient and readable'. After he was done, he saved the file and gave it back to the younger programmer, who was ashen-faced.
>
> 'That... My code is gone!' said the younger programmer. 'But you have it in version control somewhere, right?' asked the elder. 'N.... no.' was the reply. 'Well then,' said the older, 'now you've learned two lessons.' "
>
> — *Dan Udey* [3].

Having this safety net is also a very good incentive to experiment. As mentioned in the anecdote the older programmer, assuming that the younger was using version control, felt free to delete most of the code and rearrange what remained. If the younger had been using version control then they could have simply committed this major change on a separate branch and checked out their prior work to compare the two.

The main Version Control System (VCS) used by engineers is likely to be Subversion, this has been around a long time and is one of the best open source centralised VCSs. However a new generation of VCSs are coming out known as Distributed Version Control Systems (DVCS), these offer a completely different way of looking at version control.

1) Centralised VCS: Centralised VCS is still definitely the most widely used type of VCS, with Subversion one of the most widely used Centralised VCSs. The centralised VCS paradigm is centered around having a single server that stores the entire history of the project. When a developer checks a revision out of this server the content of the files at that revision is transmitted to him. The developer then changes the files and checks it back in to the server. As long as no one else has changed the files during this time the check in succeeds and the server stores the new versions of the files in a new revision. If someone else has changed one of the files then the check in fails and the developer is notified. They then have to update their working copy, this will pull down the latest versions of the files and leave the files that have changed in both places in a conflicted state. The developer will then need to merge the two files and check the updated version containing both sets of changes in to the central server.

This merging of the two sets of files in the developers working copy is one of the major downsides of a centralised VCS, at this point the developers changes have not been committed. If they screw something up when attempting to merge they have no safety net beneath them to fall back on.

2) Distributed VCS: DVCS are rapidly gaining acceptance in the software development community. The two most common DVCS are Git and Mercurial (Hg), there are a few others that are used quite a bit as well like Bazaar and Bitkeeper, however Git and Hg are definitely the most used.

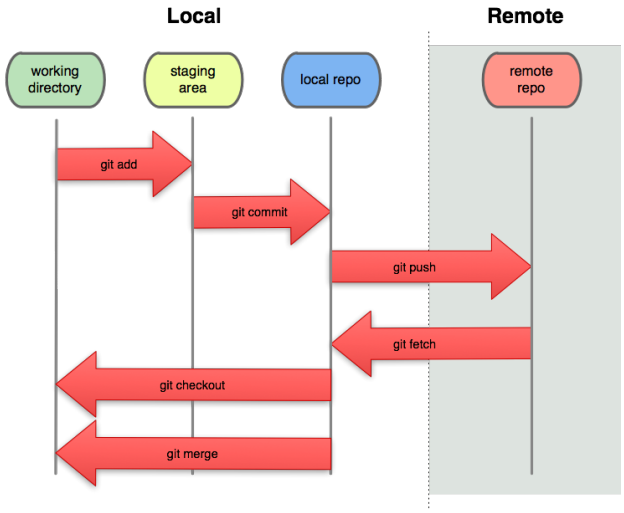

Fig. 1. Local and Remote operations in Git. [4]

a) Git: Git was originally created by Linus Torvalds for use in the Linux Kernel project. The major design goals for it were:

- Take CVS as an example of what not to do; if in doubt, make the exact opposite decision.
- Support a distributed, BitKeeper-like workflow.
- Very strong safeguards against corruption, either accidental or malicious.
- Very high performance.

(CVS was one of the original Centralised VCSs, pre-SVN).

b) Mercurial: Mercurial was created by Matt Mackall at around the same time as Git with the same planned usage, namely the Linux Kernel project. It wasn't chosen for this, but is still widely used for a lot of software projects.

The major difference between DVCS and traditional VCS is the lack of a central server. Every clone of the code is a full repository by itself. Figure 1 shows which Git operations are local to the developers machine and which are remote. All committing, checking out and merging are happening locally, the only remote operations that happen are passing changesets between different repositories. This provides a few major benefits:

- Fast access to old revisions. Since the repository is hosted locally it is only hard drive access time slowing it down instead of network access times.
- Always accessible. Even if you don't have internet or the central server is down you can always commit to your own repository.

Generally in an organizational context a central server will still be used with the DVCS system. This way everyone has a single source so they can keep their code tightly integrated and reduce the headaches introduced of their repositories diverge too much.

Comparing the workflow mentioned above on a DVCS it would go as follows: The developer updates their clone of the repository to the latest version. They then do the changes to implement whatever feature they are working on. These changes are committed to their local repository, this always succeeds since they are the only ones with access to it. They then push this changeset up to the central server. If no-one else has pushed this succeeds and the central server will be at the same version as their repository. If someone else has pushed to the central server then they will have to pull the changeset and merge it into their repository. One of the key differences at this point is that their change has been committed and can trivially be recovered if something goes wrong. Once they resolve the merge they create a new merge commit that has both their change and the other change that was pushed to the server as parents. They can then push this change up to the central server (assuming no one else has pushed during this time).

B. Unit Testing

As mentioned earlier Unit Testing is currently seeing a major increase in use in forward-thinking software development companies, mainly because of evangelical Agile development proponents (especially from the Ruby on Rails community). Unfortunately despite this increase for traditional software development, the uptake in embedded development projects has been a lot slower.

The basic premise of unit testing is that if you verify that all parts of your system work as intended, then the system as a whole will work as intended. To do the verification you write a lot of small *unit* tests to verify minimal sections of your code. By ensuring that all code you write is tested by multiple unit tests you can be confident that the code performs as you expect.

Of course this isn't the same thing as being right, unit tests only verify that the code does what the test says it should. To ensure the code does what it should do you need to validate your tests. This is most commonly done informally, the test developers know the intended outcome and write the tests with the codes final purpose in mind. For more critical systems an external validation can be performed using a method such as modelling. Our attempts at modelling one of the critical sections of our system can be read about in *Safety by Design for the Mariokart System* by Simon Richards [5].

The big problem with unit testing an embedded system is the very low level of abstraction. One of the big reasons it is so popular in the Ruby on Rails community is because of how easy it is to write tests for Ruby thanks to its very high level of abstraction. For embedded development you are likely going to want to run a few different testing layers; one testing the very low level libraries to ensure the registers are being accessed correctly, one testing mid level libraries such as character or LCD displays to make sure they're calling the low level libraries properly and one testing the actual application code.

C. Continuous Integration

Continuous Integration (CI) is another software development practice highly pushed in the Agile development

community. The main goal behind CI is to minimize the time between an error being introduced in the code base and the error being detected and fixed. This is achieved by having developers continuously integrating their changes and verifying the integrated code via an automatic test. To ensure this is happening a few key steps have to be taken; testing the code has to be almost painless, it must be easy for the developers to merge their work back into the master development branch and if a bug is ever introduced the developer responsible is not allowed to ignore it.

Ensuring that testing the code is almost painless can sometimes be a big task. Especially if this was not a priority at the beginning of development, build times and test suites have a tendency to expand out of control very quickly. This is absolutely necessary for CI to work well though. To keep the error detection time as small as possible you really want all developers to be running the test suite after every change, so really a maximum run time of two to three minutes is required. For large systems this can be achieved by splitting the test suite up, you just need to be careful to ensure all tests relating to any change will be run as part of that section of test suite.

Continuously integrating developers code requires having a single *master* development branch. Whenever a developer decides to do some work they will grab the latest version of this branch and start their work form there. Because their team is performing Continuous Integration they can be confident that the master branch will be in a working state, or if not that whoever broke it knows and will have a fix pushed up to it within a few minutes. They will then write a test for the feature they are going to be implementing, followed by the code to make the feature work. This will be repeated until they have finished implementing the feature. If for some reason this is a very large feature that is going to take more than a day or two they will try and aim for a few checkpoints consisting of just a few hours work. At each of these checkpoints they will merge in any changes that have happened to the master branch and ensure that none of them introduce any problems with the feature they're currently implementing. It is this integration of the work other developers are doing multiple times through the day that gives Continuous Integration its name. Once the feature is finished it is merged back into the master development branch and pushed out for other developers to use.

Once a feature has been merged into master other developers will be pulling it down and integrating it into their current work. If there is a bug in it then it is going to affect a lot of people. For this reason most teams using CI use a Continuous Integration server. This is simply a server set up to continuously pull any change to the master branch, build it, test it, and notify the developers if it fails. In this way if someone accidentally commits broken code the CI server will detect it and inform them within a few minutes. At this point the developer has to make getting a fix out their highest priority, if they can't see some way to fix it straight away they should revert their merge and spend time working out what broke it.

III. Application

This section will explore the application of the aforementioned software engineering techniques to the development of the software used for Mariokart.

A. Version Control

The version control system used for this project was Git. The reasons for choosing this over something else like SVN or Hg were:

- DVCS are the way of the future. There is almost nothing that SVN does better than Git or Hg, the few things it does are niche features such as being able to checkout just a sub folder in the repository.
- Two of the developers on the project had previously used Git and were using it daily for many other projects.
- Github provided us with many very useful features such as a Wiki to use for documentation and project management.

This turned out to be a very good choice, some of the major positives were:

- The wiki, throughout the project we recorded details on our decisions on the wiki. This has proven invaluable now when looking back we have to figure out details such as why exactly we chose the SAM7XC.
- Painless branching and merging. This was very helpful at a few points during the development; at one point it was decided that the Atmel CAN library was untrustworthy and would require a rewrite. While that was going on in a different branch the rest of the developement could continue on in the main branch. Once the module was re-written it was simple to merge it back in and change the few cases where the interface had changed. Also, when we were coming up to give a demonstration it was easy enough for each member to branch off and write their demo code in their own branch, this ensured that any changes they had to make to ensure the demo went smoothly could be segregated.
- CIJoe integration, this will be detailed more later, but the combination of Git and Github made it extremely simple to set up CI Joe.

B. Unit Testing

Unfortunately this was not implemented in our project. The possibility of implementing it was explored, but the lack of time prevented it from continuing.

C. Continuous Integration

Continuous integration was implemented using a piece of software known as CI Joe. This is a continuous integration server designed to be as simple as possible to setup. You provide CI Joe with a git repository and a command to use to test the project and you are done. In our case since we did not have the time to set up a unit testing system we just had CI Joe building the project to detect any compilation errors or warnings.

References

[1] D. L. Parnas, "Software engineering programs are not computer science programs," *IEEE Software*, vol. 16, no. 6, pp. 19–30, 1999. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=805469

[2] H. V. Jenkins, "Embedded hardware design for autonomous electric vehicle," 2011. [Online]. Available: https://raw.github.com/team-ramrod/mariokart/master/Documentation/ScientificReport/Henry/report.pdf

[3] Dan Udey, 2008. [Online]. Available: http://stackoverflow.com/questions/132520/good-excuses-not-to-use-version-control/135002#135002

[4] Scott Chacon, 2008, used under permission of MIT license. [Online]. Available: http://whygitisbetterthanx.com/

[5] S. Richards, "Safety by design for the mariokart system," 2011. [Online]. Available: https://raw.github.com/team-ramrod/mariokart/master/Documentation/ScientificReport/Simon/report.pdf