# Background and Related Work

## Introduction

This chapter is to explore the literature that has been written surrounding the topic of low-level programming. More specifically: ways in which programming should be taught; how this form of programming can carry forward into other aspects of life, and [ANOTHER ONE]. In addition, discussion takes place with regard to how this existing knowledge and literature will have an impact on my project, and to get the most of out of the time and resources I have available.

## Teaching Low Level Programming

### Why Teach Programming

It is well known that computer programming is an incredibly difficult thing to do and is not simply one subject. It is incredibly broad and has many transferable skills. This is because is forces people to think from a point of view of no previous knowledge, and a with a completely literal mindset. One example of this is multiplication. This is a very simple task for a human to achieve, but a computer cannot perform multiplication natively. Instead, the computer must add numbers a given number of times for example,

2 \* 3 -> 2 + 2 + 2.

This is an incredibly simple operation, but it forces the programmer to really think about everything in absolute depth. Some of these transferable skills will undoubtably be: *reasoning skills, problem solving, and self-efficacy in mathematics*, and this is exactly what was investigated by Psycharis and Kallia (2017) in their paper on the effects of computer programming on high school students. They tested this by having two groups of 33, the control group being taught *‘Development of Applications in Computer-Programming Environments’*, and the experiment group was taught Chemistry and Electrology. In addition, both groups had the same mathematics and physics courses in common. They had the same material delivered, and by the same teacher. The results of the study showed a significantly higher level of self-efficacy in mathematics, problem solving and reasoning skills, but unfortunately *the difference with the control group though was not statistically significant*. Although, it should be noted that there was a higher mean score in the group with the computing integrated in their syllabus. As such, while there was “officially” an inconclusive conclusion drawn, I think that is it sufficient evidence to prove the case that there is a notable improvement of mathematics and physics from the learning involved with computer science. I think that this elevates the importance of my project in teaching people low-level computing in a new way, as it is shown to have a positive affect on other subjects. This is excellent motivation for me to add as much as possible into the learning aspect of the project, and ensure that the problem solving and reasoning skills are being addressed in order for the impact of the project to be its greatest. I will investigate further into the methods of teaching low level programming, and ensure I link back the materials I am to deliver to broader topics, in the hope to enrich the users of my project in ways more than just an understanding on how the computer operates.

### Best Ways to Teach

Sometimes for good reason, low level programming is widely discounted when it comes to teaching computer programming. This could perhaps be the way in which it is delivered and the response to the material. Certain approaches such as a full understanding of the tools needed, and full demonstrations are mostly forgotten about. “Suppose you attend a course in cabinet making. The instructor briefly shows you a saw, a plane, a hammer, and a few other tools, letting you use each one for a few minutes. He next shows you a beautifully finished cabinet. Finally, he tells you to design and build your own cabinet and bring him the finished product in a few weeks. You would think he was crazy!” Gries (1974). This is a great example of the parallels between software and working on physical materials. It is important to note that in programming, we are presented with tools, and if not used correctly, can waste time, and cause vast amount of frustration. Gries continues to describe ways in which to teach programming correctly, and but the main points to take his paper are to: *Understand the problem, devise a plan, carry out the plan, and look back*. With these steps, one can “*develop a level of discipline which most programmers don’t have”* and transfer these skills to other aspect of one’s life, such as *reasoning skills, problem solving, and self-efficacy in mathematics* Psycharis and Kallia (2017). The list of best practises to follow set out by Gries is a great way break down a problem, and I will very much look to replicate and encourage this when developing my learning materials and programming language. I will also use this as motivation to fully teach every tool that I am giving to the user of the product. The beauty of low-level programming is that each command one does one thing, and it is building these commands into layers and layers of functionality that allows users to accomplish amazing and complex things. This is the joy I wish to be able to communicate to the user of project, and to be able to facilitate this learning to any ability.

Furthermore, while providing the correct foundation to the learning about the tools used in programming, it is also hugely important to cater for the experience level of the user. An advance programmer who has programmed their whole life will not be interested in declaring variables, or how to use a for loop; in contract, a novice programmer will become overwhelmed and put off if they are faced with advance object orientation, or nested recursion. Dreyfus et al. (1987) states that there are 5 steps from novice to expert: *novice, advanced beginner, competent, proficient, and expert.* It is imperative that each of these stages are be catered for. The journal goes on to explain in much more detail about the transition from being a novice programmer, all the way to becoming an expert, and most importantly, how the thought process changes over this time. Thought process is a huge factor in my project, as all I want to do is enable people to fully understand what is going on behind the screen in front of them. This again is where the use of low-level code comes into its own. As it is so close to the hardware it is being run on, it is not a case of “how to declare a variable” it is more, “how the variable is being declared”. By rearranging this initial question, we are able to create a new base level for anyone in the demographic to begin or move ahead if they already know. As such, the one material can cater for all stages. This way of thinking on my behalf will be reflected onto the user of my project, by fully understanding the material they are reading. I must, however, carefully write the documentation to enable anyone to understand, and have fun with.

Another consideration is to look a mental model first approach. This is whereby the actual programming comes very much second to the thought process behind what is going on. Robins et al. (2003) discuss that *programs are usually written for a purpose,* and that a *mental model of this problem domain must precede any attempt to write an appropriate program.* This is also confirmed by Deek et al. (1999) who *describe a first year computer science course based on a problem solving model, where language features are introduced only in the context of the students’ solutions to specific problems.* A study was undertaken to test this theory of using a mental model before starting to write any code, by Xinogalos (2014), the results of which showed that 40% of the 50 students involved in the voluntary study answer “very much” to “How much did the use of a pseudo-language help you in your introduction to programming?”.This mental model approach, whereby the students must first gain an understanding, not only of the problem they need to solve, but also to apply the limitations present in the tools they have available to them is great. By having this plan, and allowing the use of actually programming based solely off of the mental model they have built is a great way to allow the students to think how the computer is thinking. This also backs up Gries (1974) who states some rules to software development: *Understand the problem, devise a plan, carry out the plan, and look back.* This mental model approach incorporated the initial understanding of the problem, and devising a plan, in one. It forces the linear approach to abiding to the rules set out by Gries. As stated, these rules laid out by Gries are ones that I wish to reflect and implement into my project. I am looking to encourage the use of Deek’s mental model approach when thinking of low-level concepts, as dry running the programs in your head is an incredibly useful way to generate a simple layer of logic which is to be built upon throughout the implementation of the problem’s solution.

### A practical approach

Low level programming is all around us, and despite this, is widely forgotten about. It is the only way in which we interact with a computer, and it is the fastest way to run any program on a computer. Low level programming is a dying art. This is mainly due to high level languages and structures taking the leading role in education and software development. They are more tailored to how people think and so are much simpler to pick up. Despite this, low level programming is still incredibly relevant, now more than ever, and should not take a back seat. It is not until late education that we are taught about high level language being translated into something which can be understood by the computer being used. In a study undertaken by Smith and Webb (2000), they looked at how *meaningful learning* requires pre-existing knowledge to build upon. As we are not taught how the computer actually operates, when the topic of low-level programming emerges, the learner is forced to memorise information without proper understanding, and this will leave one’s mind very quickly. Their solution to this was to create a *glass box* interpreter based upon C, which would show a novice user exactly how the compilation process is undertaken. This is a strong parallel to my project, as I too wish to create this *glass box* model of an interpreter. The difference is that my *glass box* will be based on a custom programming language, as opposed to an existing language such as C. I think that I am likeminded to these researchers, and am out to solve the same problem. Their journal goes on to explain a great deal about the functionality of their *glass box* implementation and I am going to investigate drawing parallels to these functionalities, this is because the results of the study when introduced to real students was very positive, and they are features which I believe will improve my current design greatly. I spoke about some of the features in prior documentation, but this will add context and proof that this idea will work and will yield results. While the results from this study have been overwhelmingly positive, the way in which the responses were gathered was via a multiple-choice questionnaire. While these are quick and easy to setup and get results fast, there are a few issues with this form of feedback gathering; some people may simply guess the answers and still come out with a majority of correct answers.

In contrast to having a practical looking into the software side of things, one of the advantages of low-level programming is that is very close to the hardware that the code is being run on. This gives the opportunity to see the results of the code written, very easily. Microcontrollers (or µControllers) can have code embedded directly onto them and be used to create advanced circuitry that can be fun to make and see the results immediately. In recent years, this technology has become extremely affordable and easy to use in a DIY format. *Modern embedded C compilers employ built-in features for keeping programs short and manageable and, hence, speeding up the development process Bolanakis (2017)* The difficulty here, however, is that these built-in functions will not be universal, and very specific to the hardware being used, while the language I am using is extremely specific to my project, I wish to draw as many parallels to other uControllers, so the skills will be transferable. This approach is called *Code More to Learn Even More, and directs the reader toward a low-level accessibility of the microcontroller device Bolanakis (2017).* This “motto” describes the way in whichlow-level code (certainly for educational purposes) should be written. This way, the user will have a full understanding as to what is going on. The code that is actually written is incredibly simply, and can act as the pre-existing knowledge that built upon, as described by Smith and Webb (2000). It is what the user will do with this foundation knowledge that is important, as with low-level programming, there is very rarely one approach to a problem. This forces the user to really think about the problem, and once it is broken down into small enough pieces, it is completely manageable. This is the mindset that I want to be able to bring across in my project, if anyone using my language can begin to think in that way, then I will consider it a success.

## Transferable Skills

### Modern-day low-level programming

employment of today’s flexible and low-cost Do-It-Yourself (DYI) microcontroller hardware Bolanakis et al. (2009)

### Bridging the Gap Between Hardware and Software

Yet

### Application of µControllers, and low-level language uses

## Problem Addressing

## Summary

Write a short summary at the end of this chapter. This is not your conclusions, that will go at the end of your report. This is a summary of the chapter. Have a look at the examples on Blackboard. Do not start this section with ‘In summary’, it is obvious from the heading what it is.

Once you have finished this chapter put it aside for at least 48 hours, then come back to it. READ IT OUT LOUD to yourself, this is a good editing technique.

Then, ask someone else to read it for you.

Get some information feedback from your supervisory team. Make sure you do this with plenty of time before the submission date.

References

Bolanakis, D. E. (2017). Microcontroller Education: Do it Yourself, Reinvent the Wheel, Code to Learn. *Synthesis Lectures on Mechanical Engineering*, *1*(4), 1–193. <https://doi.org/10.2200/s00802ed1v01y201709mec009>

Bolanakis, D. E., Evangelakis, G. A., Glavas, E., & Kotsis, K. T. (2009). A teaching approach for bridging the gap between low-level and high-level programming using assembly language learning for small microcontrollers. *Computer Applications in Engineering Education*, *19*(3), 525–537. <https://doi.org/10.1002/cae.20333>

Deek, F. P., Hiltz, S. R., Kimmel, H., & Rotter, N. (1999). Cognitive Assessment of Students’ Problem Solving and Program Development Skills. *Journal of Engineering Education*, *88*(3), 317–326. <https://doi.org/10.1002/j.2168-9830.1999.tb00453.x>

Dreyfus, H. L., Drey-fus, S. E., & Zadeh, L. A. (1987). Mind over Machine: The Power of Human Intuition and Expertise in the Era of the Computer. *IEEE Expert*, *2*(2), 110–111. <https://doi.org/10.1109/mex.1987.4307079>

Gries, D. (1974). What should we teach in an introductory programming course? *Proceedings of the Fourth SIGCSE Technical Symposium on Computer Science Education - SIGCSE ’74*. <https://doi.org/10.1145/800183.810447>

Psycharis, S., & Kallia, M. (2017). The effects of computer programming on high school students’ reasoning skills and mathematical self-efficacy and problem solving. *Instructional Science*, *45*(5), 583–602. <https://doi.org/10.1007/s11251-017-9421-5>

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and Teaching Programming: A Review and Discussion. *Computer Science Education*, *13*(2), 137–172. <https://doi.org/10.1076/csed.13.2.137.14200>

Smith, P. A., & Webb, G. I. (2000). The Efficacy of a Low-Level Program Visualization Tool for Teaching Programming Concepts to Novice C Programmers. *Journal of Educational Computing Research*, *22*(2), 187–215. <https://doi.org/10.2190/n0vv-0p48-xj9g-f8wv>

Xinogalos, S. (2014). Designing and deploying programming courses: Strategies, tools, difficulties and pedagogy. *Education and Information Technologies*, *21*(3), 559–588. https://doi.org/10.1007/s10639-014-9341-9