# Background and Related Work

## Introduction

This chapter explores the literature that has been written surrounding the topic of low-level programming. Three themes relevant to this project will be discussed: the ways in which programming should be taught; an overview of the concepts and techniques within the low-level programming domain; and the application of microcontrollers. In addition, this chapter considers how this existing knowledge and literature overall will have an impact on this project, and to get the most out of the time and resources available.

## Teaching Low Level Programming

### Why Teach Programming

It is well known that computer programming is a difficult thing to do and is not simply one subject. It is incredibly broad and has many transferable skills. This is because it forces people to think from a point of view where they have no previous knowledge, and 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, consider the equation:

This is a simple operation, but it forces the programmer to analyse everything in absolute depth. Some of these transferable skills will undoubtedly 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 being 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, no discernible difference was found in the performances across the two groups. Although, it should be noted that there was a higher mean score in the group with the computing integrated into their syllabus. As such, while the study was categorised as inconclusive, it could be argued that there is sufficient evidence of notable improvement in mathematics and physics from the learning involved with computer science. This elevates the importance of this project in teaching people low-level computing in a new way, as it is shown to have a positive effect on other subjects. This is excellent motivation to add as much as possible into the learning aspect of this project and ensure that the problem-solving and reasoning skills are being addressed for the impact of the project to be its greatest. Further investigation into the methods of teaching low-level programming will therefore be valuable in the design phase of this project, to ensure that the educational materials included are linked back to broader topics. This will help users of the project expand their knowledge beyond simply how the computer operates.

### Best Ways to Teach

Low-level programming is widely discounted when it comes to teaching computer programming, although sometimes for good reason. 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. Gries (1974) illustrates this point with an example case of cabinet making. In his scenario, he describes an instructor letting students use each of the tools required for cabinet making for a few minutes in isolation. While the students are now knowledgeable in all of the component skills they need, none of the students could be expected to apply this knowledge to fully constructing a cabinet on their own. 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 amounts of frustration. Gries continues to describe ways in which to teach programming correctly, 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 aspects of one’s life, such as reasoning skills, problem-solving, and self-efficacy in mathematics Psycharis and Kallia (2017). The list of best practices to follow set out by Gries is a great way to 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 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 the project and to be able to facilitate this learning to any ability.

Furthermore, while providing the correct foundation to learning about the tools used in programming, it is also hugely important to cater to the experience level of the user. An advanced programmer who has programmed their whole life will not be interested in declaring variables, or how to use a for loop; in contrast, a novice programmer will become overwhelmed and put off if they are faced with advanced object orientation or nested recursion. Dreyfus et al. (1987) state 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 being 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 to 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 at 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. Having this plan, and allowing the use of programming based solely on the mental model they have built is a great way to allow the students to think about 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 abide by 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 that 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 like-minded 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 set up 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 microcontrollers) 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 microcontrollers, 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 which low-level code (certainly for educational purposes) should be written. This way, the user will have a full understanding of what is going on. The code that is actually written is incredibly simple and can act as the pre-existing knowledge that is 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.

## Low-Level Programming

### Low-Level vs. High-Level

Low-level and high-level programming languages are very much different kettles of fish and have very little in common. As such, it is impossible to say which is “better”, nor is this the goal of this section. With that said, there are pros and cons to both and different uses of the types of programming. These are the differences that this section is out to explore. Frampton et al. (2009) look to demystify the magic involved with programming. They state that “The power of high-level languages lies in their abstraction over hardware and software complexity, leading to greater security, better reliability, and lower development costs” This is an incredibly true statement meaning that the low-level programming languages have direct control over the hardware that is connected to a given system, and the power of the high-level languages are all based off utilising this power. The issue is that this power in abstraction removes the willingness of the compilation process to be seen by the programmer. This is shown by their later statement “opaque abstractions are often show-stoppers for systems programmers, forcing them to either break the abstraction or more often, simply give up and use a different language”. Here, they are explaining that as high-level programming is built upon low-level instructions, only commands that are deemed of use by the people making the high-level language, can be used by the developer. In some situations, the developer will need to do things that are simply impossible with a given programming language. This may force the developer to either break the abstraction (either by editing the high-level language or by writing raw assembly) in order to achieve what they need to. This is incredibly time-consuming, and expensive due to the specialism required to achieve this. Failing that, in some cases, the programming language does not and will never be able to do the operations required of it, and an entirely different language will be required to be used. The opacity is something that this project is out to address. The interpreter needs to be able to be completely transparent and explain each of the decisions being made. This is an essential skill for people who want to become specialised in this field.

In contrast, while the use of low-level programming is incredibly powerful, and can enable the developer to do almost anything with the hardware in the system, it is also the most difficult type of programming to do. This is the reason high-level programming was created: to make programming easier for humans, while also enabling computers to execute the instructions being given to it. This problem has been recognised and was addressed by Métrailler and Mudry (2015) who recognised the advent of off-the-shelf programmable embedded systems such as Arduino, which all are aimed at educating people how to use hardware, but not necessarily low-level programming. As such, some limitations exist, and this is something that Métrailler and Mudry look to rectify by building a block-based programming language being compiled directly to incredibly low-level assembly/machine code. This eliminates the need to learn a language such as C/C++ which contains extremely difficult concepts that the young demographic of these products will be spooked by and feel put off. This block-based programming was made popular by the Scratch Foundation (n.d.) who created Scratch, which is a coding language with a simple visual interface that allows young people to create digital stories, games, and animations. It is very intuitive and teaches people the basics of programming while doing something like creating a game, or animation, which (for its demographic) is a fun activity. While with this project, it is not realistic to have a visual interface, the idea of using something familiar to teach another concept is something that will very much be taken advantage of. The compile will accept the custom language which will be based on the syntax from a high-level language. In addition, the demographic of this project is someone with some programming experience, and it is this audience the language will be designed for.

### Digital Logic

Digital logic is the home of all low-level programming. It is common knowledge that there are billions of transistors inside even a typical mobile phone, but the average person is not really sure what this actually means. Digital logic is a form of thinking about how electrical signals are handled by a circuit. A common implementation of this is the TTL chip which stands for Transistor-Transistor Logic. Meaning that if you open up the chip, you will find these billions of transistors. If these are added together, you can form logic gates, such as AND, OR, XOR, etc. and by grouping these together, amazing constructs such as static memory, decision making, and mathematics can be performed. It is true that all a computer is is the abstraction of smaller things to create bigger and better machinery. Half of this project is all about creating a simplified TTL architecture to run commands on. This is a similar concept to what Rodriguez (1994) has investigated with his very oversimplified architecture. This too is very much an educational tool, and the results of which had very positive results, and due to its simple construction, was cheap and able to be used by any student that wanted it. Rodriguez did show some of the issues from his study, however. The main areas of negativity are the complexity of assembling the board, and general poor implementation. Apart from this, it overall shows a great result and encourages true promise to the project, which draws parallels to Rodriguez’s designs in that the project will also be cheap in construction, but will also learn from the mistakes made here, and focus on being simple to pick up and use.

### Relevance of Low-Level Programming

Low-level programming is still used a great deal in the modern-day, it is just a lot more specific than other higher-level languages which are easier to learn, and so more people are able to enter the industry with this experience. Some applications, however, will still require the experience of low-level programmers to create, from operating systems to general system architecture. While it is the norm to look at languages such as C++ and C# to create these types of applications, some hardware will still require intervention from languages like assembly to operate efficiently and correctly. The mere existence of recent books in assembly is a clue to this. Examples include Modern X86 Assembly Language Programming, Kusswurm (2014) and Assembly Language Step-by-Step: Programming with Linux, Duntemann (2009). These books were created to teach modern applications of assembly language, and most importantly, on modern machines, and architectures. Kusswurm’s book is actually written for an x86 architecture, which is the main architecture for 32-bit based operating systems (the name is from the 32 bit Intel 8086 microprocessor). These books are incredibly good reads, with a great deal of information about the usage of assembly in the modern-day. They are primarily based on learning assembly (not to a young audience), however, some of the techniques here will be perfect draw parallels to for this project. Furthermore, it would be pointless to make a completely new version of the assembly syntax, as such, this project will not look to reinvent the wheel, but to replicate the basic syntax, but simplified slightly for the target demographic.

### Applications of Low-Level Programming

Any computer or device uses low-level programming, after all, low-level programming is not a language in itself, it is more of a concept. Machine code is the only language that computers can use. Take Windows 10 as an example. While it is unfair to call it one piece of software, it is in fact layers of code all working to manage the PC’s resources. It can be thought of as a large piece of software called an operating system. It was created with the use of multiple languages, the kernel (the lowest part of the OS) was developed in almost pure C (a very low-level language). OS Today (2019) wrote about this and explained the primary use of C and assembly is for such a purpose.

More specifically, low-level languages such as C or Assembly can also be used for microcontrollers. They have the ability to directly access the hardware peripherals under the microcontroller’s control. This is because the languages allow for the actual manipulation of the architecture itself. This has advantages more than just being able to use all of the software, but it also improves the performance of the code to be run as the logic is not compiled or interpreted by another person’s software. It is your logic that is being run on the computer. When it gets to this low level, every clock cycle can add complexity and thus increase the time it takes for the program to run. LoopPerfect (2019) has a great explanation of this, the outline of which is that the reason higher-level programming is less performant is that generic low-level commands are required to be able to compile any generic program. Microcontrollers are a fantastic tool and are a very common use of low-level programming which this project aims to help facilitate the learning of.

## Microcontrollers

### Understanding

A picture containing graphical user interface

Description automatically generated The point of microcontrollers is not to be a fully universal tool for every application, but to be incredibly specific to the task at hand. A microcontroller used in a vending machine should not be expected to have some software changes and run in a NASA satellite. As such, it is imperative that the correct controller and hardware are chosen for the job. The perks of having such a specific system are that they will be cheap to create and manufacture in bulk, meaning that the technology can be more affordable and not have unused features. For example, if you buy the biggest and most expensive multitool out there, but you only want to open the occasional package, you won't use 90% of the other tools in its arsenal. It will be too expensive and large for the job it needs, instead, purchase a single retractable knife, it performs one job, it is cheap to purchase, and gets the job done just as well as the expensive one, if not better. A microcontroller is the knife in this example, a cheap and cheerful solution, and with a few subtle tweaks and additional hardware, can be the solution to any problem to any problem. The ability to chose which controller to use, and which hardware to use as the peripherals is the difficult part, and much research should be put into selecting the right tools for the job, as with any solution. Yılmaz et al. (2017) wrote an article to explain exactly this. They do this primarily by comparing various different types of controllers, peripherals, and architectures. The most interesting and relevant to this project however was the comparison of development boards:

Figure

These are great boards, mainly for educational purposes, but can also have industrial applications. While some of these boards are very well known and used a lot in education, they do very much hide a lot of the ins and out of what they are actually doing, but such is the nature of higher-level architectures. The raspberry pi, for example, is not a microcontroller, as it has a full operating system, “a computer” is a much better description of what it is. The wide range of technologies available in this area is made very obvious from this study. While my project is not aimed at advising the user about which microcontrollers and hardware to use, I do feel as though understanding the limitations given by low-level programming will enable them to make their own decisions and be able to carry out their own research as to what is available to them to solve the problem they are out to face.

### Applications

Microcontrollers are processors which have the capability to be used on a wide range of devices, from calculators to vending machines. While such devices like this on the outside are very much a hardware product, they would be completely useless if it were not for the software burned into the microcontroller itself. This is demonstrated by Tanshi and Bello (2014) who show the vast amounts of applications of the µController. The article is mainly focused on the implementation of a basic calculator using a µController as the core. As such, a big focus of the paper was about the hardware requirements for such a device, and this is where the power and flexibility of microcontrollers come into play. The µController is only as powerful as the peripheral devices it has available to it. Here, Tanshi and Bello are using hardware to create a user interface for a calculator, but could just as easily make one for others such as automatic transmission cars, fuels pump meters, washing machines, digital cameras, DVD’s, mobile phones, vending machines, etc. Just from this, it is clear that microcontrollers are all around us, and they will only ever become more powerful, flexible, and most importantly, still in demand. This demand drives up the need for people who are able to program such devices in the future. People who are used to only programming in a high-level environment will simply not have the know-how to make devices like the ones listed by Tanishi and Bello operate. One might argue that modern microcontrollers should have the ability to use high-level techniques, but this adds complexity, and cost. This thus size to the units themselves and begins to degrade the very point of having such a low level and specific device. The goal of this project is to explain to users how low-level programming can be achieved, thus helping to develop the next generation of low-level programmers and to spark interest in this dying art.

In addition to consumer products being powered by microcontrollers, research projects have been undertaken by countless academics and companies to better their products or research with the use of microcontrollers. This is due to the relevance, availability, and simplistic nature of the microcontroller which shows a true reflection on how good the technology is. Shao (2006) created an Improved Microcontroller-Based Sensorless Brushless DC (BLDC) Motor Drive for Automotive Applications. He developed a cheap and effective new way to monitor the back EMF produced by electric motors which can more accurately monitor the motor positioning, while also providing overcurrent protection. Zwerg et al. (2011) created a new form of microcontrollers for energy harvesting applications. It draws only 0.000082Amps (82µA) which makes this technology perfect for applications in energy harvesting where every nano amp drawn, counts. Babiuch et al. (2019) Even take a look at the applications of the internet of things with embedded technology, which is something that everyone will soon be using, and most of the time without the user's knowledge. As shown, there are countless applications for this type of technology, which are aiding in the progression of many different industries that are used by all types of people all over the world. The applications will only continue to grow with the development of new technologies and industries that we don’t even know about today. As such, it is imperative that people remain clued up in this area to keep the up-and-coming technology developing. This project is aimed to achieve this education upkeep, as low-level programmers are becoming few and far between with high-level technologies being easier to understand.

### Relevance

With the knowledge of what microcontrollers are, and how old the technology is, one may begin to wonder what the point of learning this old form of thinking is. With great advancements in technology, surely the use of these small and specific devices will phase out, and no one will need to know any low-level programming languages or logic. Mitchell (2020) made an argument for this being far from actual fact. His primary arguments consisted of the incredible complexity of larger, more powerful microcontrollers, along with the vast increase in price which is attached to this. He concludes by stating that the industry shows no sign of a reduction in 8-bit usage, and their low-cost nature combined with simplicity still makes them highly relevant. In addition, Pal (2017) covers some very interesting information about the reasons for using an assembly language (a very low-level type of programming). He shows how the use of assembly offers complete control over the system’s resources, direct access to hardware, a deep understanding into how the computer actually operates, complete transparency, and more. These points are extremely potent in this project, as it aims to address each of these positives for learning low-level programming. The whole point of the project is to be as transparent as possible while also providing a somewhat powerful outcome. All of the hardware will be available all of the time, meaning the user can fully grasp what everything does and will have the ability to play around with all of the parts. As shown, this technology is still rife in the modern world, despite it being old. This gives me great confidence in the project and only strengthens the fact that this is needed and there is a serious gap in the market here.

### Uses in Education

The point that microcontrollers are not taught very much in education due to their complexity, is a major driving factor of this project. “Microcontrollers account for the majority of processors produced today, yet their capabilities are seldom explored in modern computer science curriculum” This is a sad quote from a study by Margush (2006) which goes into the importance and relevance of assembly language. This quote is one thing that my project intends to address. The study by Margush continues to talk about different ways in which to compile assembly code. One example of this is for the Intel 8080 microcontroller “Windows XP platforms to write, assemble, debug, and execute Intel 8088 assembly language programs”. The issues that are raised here regard the way in which the OS allows access to low-level things like I/O and interrupts This is a theme that is shown across a wide range of assemblers (the compilers for assembly code). This problem with transparency is again a problem that prevents people from gaining a further understanding as to what goes on inside the computer. The usual phrase used is that “the code is turned into something the computer will understand”. The project will ensure that there will be complete transparency in the workings of the compiler, showing the user the output of stages as it goes and shows the final output, so the user has a complete view as to what is going on.

## Summary

This section discusses the current literature surrounding the area of education and applications of low-level programming and how the lessons learned here can be applied to this project.

The education of low-level programming has become rather limited and is barely taught at all due to the invention of high-level languages. While this is not necessarily a bad thing, it does introduce some potential limitations for advanced developers who have reached the extent of the functionality of high-level programming when it comes to fully utilising the resource of a PC; Integrating hardware into their projects; or simply executing an operation which is not possible given the constraints on the high-level programming due to its very nature of abstraction. It has been shown that there are great benefits of teaching programming and starting low-level principles will improve anyone’s literal and logical thinking which is incredibly transferable to other STEM-based subjects.

The application and usage of microcontrollers is also a primary talking point of this chapter. Their relevance in the modern day is still very much evident with almost every device in the world running some form of microcontroller with low-level software written on it. As such, the need for people who understand this type of programming will only increase. The performance of the low-level direct hardware addressing is a large driving factor for this. With hardware, for example, the process cannot simply be paused, the controller must be available to process any input it provides, or critical failure could occur.

There is most definitely a place in this world for high-level programming, and this chapter proves this, but it also analyses the other side of the argument whereby low-level languages should not be forgotten about and thought of as simply another layer of abstraction that is too complicated for anyone to understand. This would be incredibly sad, as the future of computing would be much less performant, and it would be near impossibly to acquire full implementation of the latest and greatest in peripheral technology.

To finalise, both forms of programming have a place in this world, but respect should be given to both, and not to forget about the very foundation of modern-day computing, as without it, even today, almost every modern commodity would not exist.

References

Al-Dhaher, A. (2001). Integrating hardware and software for the development of microcontroller-based systems. *Microprocessors and Microsystems*, *25*(7), 317–328. https://doi.org/10.1016/s0141-9331(01)00124-7

Babiuch, M., Foltýnek, P., & Smutný, P. (2019). Using the ESP32 Microcontroller for Data Processing. *2019 20th International Carpathian Control Conference (ICCC)*, 1–6. https://doi.org/10.1109/CarpathianCC.2019.8765944

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

Duntemann, J. (2009). *Assembly Language Step-by-Step: Programming with Linux* (3rd ed.). Wiley.

Frampton, D., Blackburn, S. M., Cheng, P., Garner, R. J., Grove, D., Moss, J. E. B., & Salishev, S. I. (2009). Demystifying magic. *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments - VEE ’09*. https://doi.org/10.1145/1508293.1508305

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

Kusswurm, D. (2014). *Modern Parallel Programming with C++ and Assembly Language: X86 SIMD Development Using AVX, AVX2, and AVX-512* (1st ed.). Apress.

Littler, C., & Littler, C. (2017, August). *Designing a Programming Language: Part II*. Kidscodecs.Com. https://kidscodecs.com/designing-programming-language-part-ii/

LoopPerfect, L. (2018, May 21). *Why are low-level languages fast and high-level languages slow?* Medium. https://medium.com/@LoopPerfect/why-are-low-level-languages-fast-and-high-level-languages-slow-9034c7e74da8

Margush, T. (2006). Using an 8-BIT risc microcontroller in an assembly language programming course. *J. Comput. Sci. Coll.*, 15–22. https://doi.org/10.5555/1181811.1181813

Métrailler, C., & Mudry, P. A. (2015). ESPeciaL: an embedded systems programming language. *Proceedings of the 6th ACM SIGPLAN Symposium on Scala*. https://doi.org/10.1145/2774975.2774982

Mitchell, R. (2020, September 2). *Are 8-Bit Microcontrollers Still Relevant?* Electropages. https://www.electropages.com/blog/2020/09/are-8-bit-microcontrollers-still-relevant

OS Today. (2019, June 22). *Question: What Language Is Windows 10 Written In?* OS Today. https://frameboxxindore.com/windows/what-language-is-windows-10-written-in.html

Pal, K. (2017, February 6). *Why is learning assembly language still important?* Techopedia.Com. https://www.techopedia.com/why-is-learning-assembly-language-still-important/7/32268

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

Rodriguez, B. J. (1994). A minimal TTL processor for architecture exploration. *Proceedings of the 1994 ACM Symposium on Applied Computing*, 338–340.

Scratch Foundation. (n.d.). *Scratch - About*. Scratch.Mit.Edu. https://scratch.mit.edu/about

Shao, J. (2006). An Improved Microcontroller-Based Sensorless Brushless DC (BLDC) Motor Drive for Automotive Applications. *IEEE Transactions on Industry Applications*, *42*(5), 1216–1221. https://doi.org/10.1109/tia.2006.880888

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

Tanshi, F., & Bello, N. (2014). A Basic Approach to Designing Embedded Systems Using a Simple Calculator and C Programming Language. *Nigerian Journal of Technology*, *33*(3), 359. https://doi.org/10.4314/njt.v33i3.14

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

Yılmaz, G., Ercan, C., Sıtkı, K., Harun, G., & Eray, Y. (2017). Understanding the concept of microcontroller based systems to choose the best hardware for applications. *International Journal of Engineering and Science*, 38–44.

Zwerg, M., Baumann, A., Kuhn, R., Arnold, M., Nerlich, R., Herzog, M., Ledwa, R., Sichert, C., Rzehak, V., Thanigai, P., & Eversmann, B. O. (2011). An 82μA/MHz microcontroller with embedded FeRAM for energy-harvesting applications. *2011 IEEE International Solid-State Circuits Conference*. https://doi.org/10.1109/isscc.2011.5746342