Embedded Development

An analysis of different languages used in embedded system applications.

By: Gabriel Stroe

# Introduction

Today’s modern world would be unable to function without the complex electronic systems that 50 years ago were just emerging as new and futuristic. Embedded systems are largely what drive this technology as things like cars, manufacturing robots, drones, digital cameras, refrigerators, measurement instruments, keyboards, mice and so many more devices all rely on special hardware and software to get them functioning the way they do. In embedded system the type of hardware used can differ across many platforms, designs, limitations and processing speed. Due to all this variance in hardware, the software used in certain applications also greatly varies. This document explores the different types of programming languages used today in embedded systems and how they compare and contrast to one another from an embedded systems viewpoint.

Before investigating the different embedded system software types, one must differentiate between it and application software. What most people see as application software is some piece of software that an end user is interacting with or is being used as a process within a traditional operating system that the user interacts with. Applications such as a web browser, PDF viewer or even a passive statistics tracker all fit this description. Now this definition does not include all applications as there is a broad range of functionality and environments that applications can have, but it does include the vast majority. For embedded system applications the general censes is that they are programs meant for very specific purpose and usually integrates directly with the physical hardware of the system. It is often described as the firmware of a particular device and typically handles its interaction with other devices. Another large difference between application software and embedded software is due to the embedded systems lack of physical resources and processing power. The lack of resources means that size of the embedded program is typically much smaller and that it rarely ever relies on other files on the system or any data from a database. The embedded application simply reads in the input from an external source or connection and processed that data and sends it back out to another externally device. Traditional application software tends to interact directly with system files as well as have a direct control of how the data or function processed is returned to the source/user.

# Assembly Language

Today, assembly language is largely just used as an intermediate language between the machine code and the high level language. However, in embedded systems it is often used for code optimization and to fix certain errors that may be unique to the system. Not all compilers compile the high level language to assembly and then to machine code (there are a few that go straight to machine code), but many of the ones used by embedded system do. This allows the embedded system programmer to modify that code before then compiling it again to machine language.

Modern compilers of all languages have evolved to create some very efficient assembly code, but typically could be optimized even further. This is due to the compiler creating some overhead in order to handle multitude of scenarios that could have be intended by the programmer. A programmer with knowledge of assembly could go in and remove these cases of overhead and reduce the number of clock cycles needed to execute the code. This type of optimization is rarely needed in desktop or mobile applications, as the number of clock cycles a program uses is not the bottleneck of its performance. But, in a system that needs to read and process real time data at an extremely rapid rate, these few clock cycles can be the difference of the results being correct or complete skewed and worthless. In addition since the since there are physical limitations to these embedded systems in terms of memory usage, processor speed and power usage, the optimizations may need to done just to get the system to work all.

There are some programmers today who will write the entire program for an embedded system in assembly. Typically the application must be small in size and it was done due to an extreme limitation on physical resource. Coding in assembly is generally much slower than using a higher level language due to high level languages representing code in a manner that is easier for humans to understand. Many do feel that as the physical technology advances and allow for much more memory and processing power with less power consumption the need to program in or even modify the assembly could disappear entirely.

Much like how many high level languages can have similar syntax to one another, different Assembly languages have very similar syntaxes to one another. Although unlike higher level languages the entire reason that different Assembly languages exist is directly due to the difference processor architypes. Each assembly language is created to be assembled by the assembler into the specific machine code that is understood by the microprocessor. This results in some small difference across the many different processor architypes. The general logical structure stays the same but differences like using MOV instead of MOVE as well as some functions unique to certain microprocessors are used. Common assembly languages used today, due once again to the current usage of that processor architype, are: assembly-ARM, generally used for mobile ARM processors; Assembly-MIPS, used in a lot of videogame consoles as well as some Apple Products; and Assembly-x86/x64, used in modern computer systems.

# Embedded C

C is presumably the most popular language used for embedded systems. This is due to its efficiency when compiled to assembly and the relatively short development time. C provides a lot of the same or similar low level access that assembly provides but with a much more human friendly structure. In addition, C can be complied directly to machine code for several compilers out there today. Lastly, C is also extremely popular because, unlike many other high level languages, C is relative small in data size which makes it easy to develop with on a system with very limited memory (even today RAM in an embedded system could be limited to as little as 2 MB).

Directly comparing C to Assembly language, one can see why most developers prefer it. C language uses a syntax that is much easier to understand and faster to write. Consider a for-loop, in C it only requires 1 line of code that is directly understandable to humans:

for(int x= 0; x < 10; x++)

One can look at the code and easily explain what it does: for a value x that starts at zero, perform the task listed below, keep doing this until x is no longer less then 10, and increment x by 1. Also all of that functionality is done in just a short segment of the code. Now let’s look at an example of an assembly for-loop:

ci ds.b 1

clr ci

loop1 (action goes next)

inc ci

cmp ci,10

jle loop1

This piece of code looks inherently confusing and seems to have a lot of little functions that need to take place in order to perform that same loop. Also each line is one very miniscule step: define storage for ci, clear ci, perform action, increment ci, compare ci to 10, jump back to loop1 if true. Clearly, it is not nearly as fluid as the C implementation.

When in an embedded system environment, the type of C that is used is referred to as Embedded C. It is very similar to the standard C language that most programmers are familiar with on the desktop. In fact, it is actually the same language with a few extensions added for specific embedded system applications, such as accessing the memory registers and IO that would not be present in a normal C application. The other major difference is that for many Embedded C applications all of the included premade libraries will not exist. A few might, depending on the development environment for an embedded system, but most don’t due to the lack of memory on the system and the vast differences between the systems. Thus the all of the helper functions and premade libraries must be created by the programmer. Many developers of system on chips (SoC), which are commonly used for embedded systems, will include some C files that store helper functions and structs. This can greatly help the programmer in developing the program for their embedded system.

While C has several advantages for being used in an embedded system when compared to assemble and other high level languages, its flexibility in certain situations can lead to a few problems. The first issue with C is not so much of an issue as it is just a very difficult to understand as a concept, pointers. Pointers have immense power and flexibility in what can be done with them but it’s a concept that few computer programmers really understand to the fullest and are often implemented poorly or lazily. The next issue with C is that it’s generally weakly typed, meaning that the program could still compile if an argument is being passed along a function is not the exact type. Weakly typed programs often lead to runtime errors that were not catch by the compiler due to incorrect type conversions. Lastly the dynamic memory allocation and fragmentation in C is very basic and not really meant for real time systems which the majority of embedded systems are. All of these issues are manageable though, which is why C is the most popular embedded programming language.

# C++

With the popularity of C++ in the application software sphere it’s a wonder why traditional C is more popular in the embedded system world. C++ is an object oriented language, meaning that if uses the idea of classes to modify and interact with what can be view as an object. This concept can actually be very useful in embedded systems as one could make each hardware module as a class and access each function within the class to enable and interact with that hardware module. In situations where you have SoC with a lot of different components available to it, this feature is immensely useful as you could just create all of the functionality of a component in a single class and then reuse that exact same code. Another benefit for C++ is it is directly based on C, it was even originally developed with a pre-processor that took the C++ code and converted it directly to C. This means that it could work on any system that can use C as a base language. It also means all of the functionality of C is also included in C++. Today there are compilers that can take the C++ code and compile directly to assembly or machine code, even for embedded systems. Also, while still having the pointer complication that C does, in C++ there are references that make it much easier to work with and understand from a programmer’s perspective. All this plus namespaces, function name overloading, operator overloading, virtual functions, templates and so many other constructs that C++ brings to the table are so very useful in embedded systems.

So with all of these benefits, why is C the most popular language in embedded systems? With all of these constructs that C++ brings to the embedded systems world comes at one large cost: Overhead. C++ has a pretty substantial overhead that it creates when compile so that it can give the programmer all of these tools. For many years embedded systems have been created with few physical resources left to spare when using C or assembly. In those type of environment there is really no place for the overhead of C++. Today there are several tools that help to “clean up” and reduce the overhead created, but C++’s history of being resource intensive it has led to many embedded developers never bothering to invest into it. There is a hope for it still, as these tools get better at reducing the overhead of C++ and as physical technology gets better and the room for larger code expands, new embedded system developers are starting to use C++ as their main language for development.

Now with Assembly, C and C++, that covers about 95% of all the embedded systems out there currently. The last 5% are languages that are used for special cases or are just starting to have compilers developed for it in an embedded system environment. There are several languages in this 5% including a dozen or so that are no longer in use today. Next this document will looked at just two of those languages: Java and Python.

# Java

Java as an embedded system language is an interesting one, as many of its original design ideas are useless in the embedded environment. For example Java gets compiled into a “bytecode” which is then interpreted at run time. This was created so the Java application could be written and compiled once, then having it being able to run on any system from there on out that had the bytecode interpreter for that system. This key feature of Java is useless in an embedded system as the code is inherently unique to the system it’s being developed for and would not be able to run on another system. In addition to that, Java’s flexibility, like C++, gives it a large overhead cost, making it a very hard sell for embedded system applications. Also, Java has a complex garbage collection, which takes up precious resources in an embedded system and really slows down the processing in a real time application. The last thing Java has going against it in embedded applications id that Java compilers and bytecode interpreters are only available for x86/x64 and ARM processors. This isn’t as much of issue today but in the early 2000s, most embedded systems processors where not of that type.

There are still some uses for Java in embedded systems. Java is an object oriented language, so it is used for specific applications where using just C or Assembly are just too unwieldly and is typically used as just an extension to the already existing C code on the embedded system. Another common use for Java is as an application within a Real Time Operating System, which is an operating system that can hold multiple applications while processing data in real time. Java’s flexibility and true Object Oriented-ness is what makes it a very good candidate as a companion to a Real time OS.

# Python

Python is an interesting language because it is the first on this list (other than assembly) that is not based in C. It is still generally an object oriented language as everything is an object, while at the same time supporting procedural and functional styles of programming. Like Java it is compiled into bytecode and then interpreted. Because of this, it is almost never used in a “bare metal embedded system”, also known as an embedded system without an operating system. The typical usage case for python is to be run in an embedded system that is typically running a very light version of Linux which is constantly just running the python application. It still has most of the same negatives that Java had when being considered for an embedded system, except that the code typically uses much less resource then Java does.

Even with this limitation of requiring an OS, Python has actually made quite a large advance in embedded systems, surpassing Java. This is mostly because the Raspberry Pi, a small and cheap SoC with a special light version of Linux running on it by default. The makers of the Raspberry Pie also promoted Python as the main language supported for the SoC, because it is very easy to learn. This in addition to the Pie being marketed as an affordable system for education, really pushed the idea that an embedded system could run an modern OS and use something like Python for all of its functionality. A concept that is very different from philosophy of embedded systems of the past.

# The Future of Embedded System Software

Where will embedded system languages go in the next few years. Today, C dominates the embedded systems platforms. But, will it stay way? Will another language gain popularity and beat out C as the new king of the embedded world? Would it even be just one language to rise, or several? These are the questions many embedded system designers are asking today. Many of these developers are younger and have grown accustomed to the newer languages such as C++, Java and Python. Many of them never even dabble in assembly and have only a remote understanding of C. In the future assembly will most likely still be used when necessary, although the number of programmers able to use the language will become even rarer. But, based on the current environment of embedded systems; it is very likely that C’s popularity will decrease.

There is likely to be 2 languages that become much more commonplace, each in their own scenario. The first will be C++. The limits of processing power and memory limitations will vanish as the hardware improves and becomes less expensive to produce. These applications will probably not use another operating system and will instead be based on the idea of a firmware approach to an embedded system where the code a programmer writes is the “operating system” for all intents and purposes. Many developers will find the object oriented-ness of C++ to be way more attractive for embedded systems that have several different hardware components or that are very modular. Also the lack of a garbage collector actually can really help C++ gain ground over Java in these usage scenarios as it will save a lot of time and resources in a Real Time System.

The next language likely to gain ground, and for the most part looks like it already is gaining a lot of popularity, is Python. Currently it is the fastest growing embedded system language even if it current only holds less than 5% of all embedded system application. It will only be used in situations where a light OS is present on the system, but its ease of use, efficiency, and versatility will likely allow it to take the spotlight in these situations. Again, as technology advances and there is more room for an operating system in addition to the real time applications on these embedded systems, the value the C brought will go down and languages such as python will gain favor.

With all this said, the future is still a blank slate. It is truly difficult to tell exactly what will happen in the next few years. Perhaps a new language like B#, which promises all of functionality of Java and C++ while still maintaining efficiency in embedded systems, will take over the entire embedded scene. Or perhaps the concept of running an embedded system without an operating system will be a thing of the past. Either way, what is almost guaranteed is that much like how faster desktop computers allowed new languages with new constructs to take the scene, the same will happen in the embedded world. In the next few years embedded systems will see a large influx of languages and functionality that was never before considered. Today is the pivotal point for software in the embedded systems world.

# Works Cited

Cellan-Jones, Rorry. "A 15 Pound Computer to Inspire Young Programmers." *BBC*. BBC, 5 May 2011. Web. 1 May 2016.

Herity, Dominic. "Modern C++ in Embedded Systems – Part 1: Myth and Reality." *Embedded*. N.p., 17 Feb. 2015. Web. 01 May 2016.

Richa. "Embedded C Tutorial: A Beginner’s Guide." *Udemy Blog*. N.p., 17 June 2014. Web. 30 Apr. 2016.

Ripps, David L. "Java for Embedded Systems." *JavaWorld*. JavaWorld, 1 Sept. 1996. Web. 01 May 2016.

Rouse, Margaret. "What Is Embedded System? - Definition from WhatIs.com." *IoT Agenda*. TechTarget, May 2009. Web. 30 Apr. 2016.

Stroustrup, Bjarne. "Embedded Systems Programming." *Www.stroustrup.com/Programming*. N.p., n.d. Web. 30 Apr. 2016.

Walls, Colin. "Embedded Systems Programming Languages | EE Times." *EETimes*. UBM Communities, 12 Sept. 2014. Web. 30 Apr. 2016.

Walls, Colin. "Is Assembly Language Best?" *Embedded*. UBM Communities, 14 Dec. 2012. Web. 30 Apr. 2016.

"What Is an Embedded System? - Definition from Techopedia." *Techopedia.com*. N.p., n.d. Web. 30 Apr. 2016.

Wilson, Ron. "Is Tomorrow's Embedded-Systems Programming Language Still C?" *SDJ*. N.p., 14 Mar. 2016. Web. 01 May 2016.