Embedded for Everyone

Improving accessibility to industry-relevant embedded systems development for hobbyists, students, and educators

Author: Nathan Jones, Master's student, North Carolina State University
Advisor: Alexander Dean, Professor, North Carolina State University

Abstract

Learning about or teaching industry-relevant embedded systems development is difficult but as important as ever before, not just to fuel this growing sector of our economy but also, and more importantly, to ensure that today's pressing technological problems are solved by a diverse group of engineers who understand the social impact of their work and can create solutions which are inclusive of many different groups of people. With the pace of technological growth, however, there exists a widening gap between the tools and materials that are available and/or suitable for educational purposes and that which are used in the industry[1]. The aim of this work is, like educational platforms such as Arduino[2], to reduce the entry barriers to embedded systems development in ways that, unlike the Arduino, are more industry-relevant. The author was able to produce the following material in service of this goal:

- this paper, describing which tools are considered "industry relevant"; how a hobbyist, student, or educator might go about using them; and what challenges they might face along the way;
- five "getting started" guides, which provide step-by-step instructions for how to set up the toolchain for, and blink an LED with, six hobbyist-friendly microcontrollers (MCUs) using 11 different toolchains;
- 14 example projects using the various MCUs and toolchains above, which include a handful of novel makefiles and shell scripts;
- two custom breakout boards for two STM32 MCUs that can be bought, fully assembled, from JLCPCB in quantities of 10 or more for $3.30 or less (at the time of this writing); and
- 12 Git repositories that contain the above design files as well as a wiki of more than 10,000 words describing how to build and understand industry-relevant embedded systems.

Much work remains, however, to address the large number of challenges facing hobbyists, students, and educators who are attempting to learn about or teach embedded systems today.

Introduction

An embedded system is a "computerized system that is purpose-built for its application"[3], and examples include everything from weather stations to mobile phones. Versatile, robust, and getting cheaper every year, embedded systems have found their way into nearly every aspect of our lives. For this reason, interest in embedded systems has grown amongst both hobbyists and educators, who are encouraging students as young as those in secondary school to learn about

[1] El-Abd, "A Review of Embedded Systems Education in the Arduino Age: Lessons Learned and Future Directions."
[2] Here "Arduino" is meant to refer to the most well-known development setup: a development board with an AVR MCU which is programmed in the Arduino IDE (which does not include any debugging capabilities).
[3] White, *Making Embedded Systems*.

this complex field[4]. To simply keep pace with the current job market would be sufficient justification for this interest and encouragement, however there is an additional reason to study embedded systems that the author finds even more compelling than simply preparing a future workforce; It is best expressed by Massimo Banzi, one of the co-creators of the Arduino platform:

> “I think that it’s important especially for kids to understand the world we live in. Clearly, if you know how to design and build things, you can affect the world that surrounds you. If you aren’t able to participate in the world of creation in the digital space, you’re left out. Somebody else is going to design your world. At some point, if there’s no innovation or even renovation in the marketplace, then one company will decide that there’s one way you do a certain thing. It becomes the only answer to a certain question, and nobody thinks about alternatives.”[5]

As embedded systems get more and more complex and become a bigger and bigger part of our everyday lives, the ramifications of allowing a small minority of engineers and commercial interests to define how those devices are built and operated could be exclusionary, at best, or perilous, at worst. A December 2018 report by the Committee on STEM[6] Education of the National Science and Technology Council emphasized, repeatedly, the necessity that “quality STEM education should be accessible to Americans of all ages, backgrounds, communities, and career paths” and noted that among those who are underrepresented in STEM fields or who have difficulty accessing STEM education are “women, persons with disabilities, and three racial and ethnic groups—Blacks or African Americans, Hispanics or Latinos, and American Indians or Alaska Natives” along with “24 million Americans who lack access to basic broadband services”[7]. The public must be able to participate in these technological endeavors and therefore must be given the tools and instructional materials to understand and experiment with these complex embedded systems; we should be co-creators in our technological future, not simply bystanders. Indeed, this is the reason that STEM education is as important as ever before, not simply because it helps make us smarter, but because it democratizes the bleeding edge of new research and development, helping to ensure that problems are creatively solved and that the social impacts of new technology are thoroughly considered. Educational tools which simplify the difficult parts of a problem are important in this regard, but insufficient to fully open the space where new embedded systems development is happening.

The intent of this paper is to survey the embedded systems development tools that are currently used in the industry, to evaluate which is most appropriate for hobbyists, students, and educators to use as tools for learning and developing, and to expand that which is accessible to

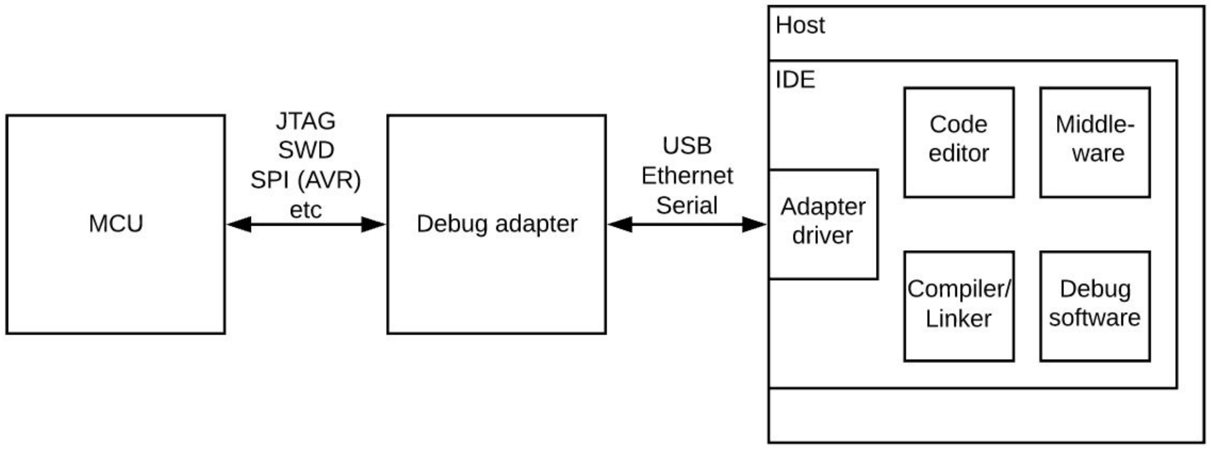
[4] Weiner, Lande, and Jordan, “What Have We ”Learned” from Maker Education Research? A Learning Sciences-Base Review of ASEE Literature on the Maker Movement”; Kurti, Kurti, and Fleming, “The Philosophy of Educational Makerspaces, Part 1”; Hlubinka et al., “Makerspace Playbook: School Edition.”
[5] Severance, “Massimo Banzi: Building Arduino.”
[6] STEM: Science, Technology, Engineering, and Mathematics
[7] Committee on STEM Education, “Charting a Course for Success: America’s Strategy for STEM Education.”

these groups by creating novel tools and training materials. The first section, “Literature Review”, will present technical definitions, including what is meant by “industry-relevant”, along with a description of the specific challenges faced by hobbyists, students, and educators in learning about or teaching embedded systems. It will conclude with the problem statement that was used for the duration of this independent study. The second section, “Finding Hobbyist-friendly, Industry-relevant Microcontrollers”, will describe the author’s search for MCUs that were both industry-relevant and hobbyist-friendly. It will conclude with a list of the MCUs that were found to meet these criteria and which has been sorted by ease of use. The third section, “Evaluating the Associated Toolchains”, discusses the toolchains available for six of the MCUs identified above, including their overall features and ease of use. The fourth section, “The Embedded for Everyone Wiki”, provides an overview of the wiki page that was written to educate hobbyists, students, and educators in the things learned throughout this study. The fifth section, “Recommendations for Hobbyists, Students, and Educators”, provides guidance to those individuals who are getting started with embedded systems based on my experience with this independent study. The final section, “Future Work”, discusses the greatest challenges facing new developers and offers a few ideas for how MCU manufacturers or open source contributors might alleviate some of those difficulties.

Literature review

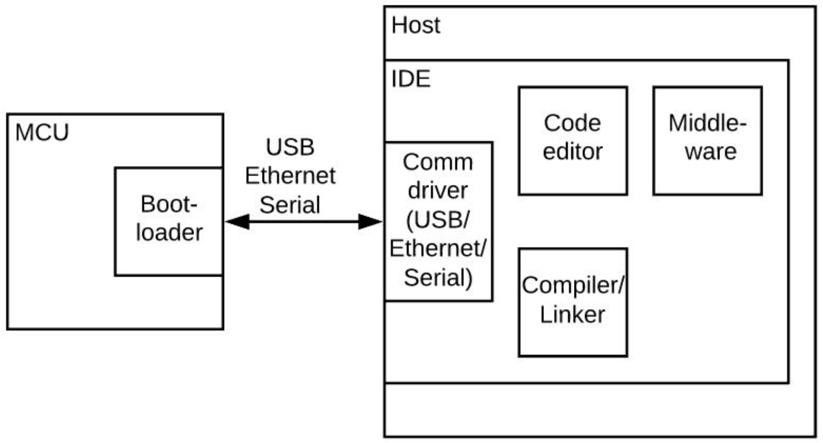
Definitions

At the heart of nearly every embedded system is a microcontroller (MCU) which executes application code written by a human developer; the application code is typically written on a more capable computing device (i.e. a modern laptop or desktop computer) and then transferred to the microcontroller. The hardware and software tools needed to convert the application code to a form that the microcontroller can execute and to load that code onto the microcontroller is called the “development environment”, the “development toolchain”, or simply the “toolchain”. The toolchain consists of six elements, as shown in the figure below, which are sometimes packaged together by a vendor to provide a complete toolchain solution.



The MCU typically only contains exactly as much digital circuitry as is needed to load and run programs, plus a few debugging features. Much of the hardware necessary to load and debug programs, however, is not needed when the embedded system is in operation and so often lives on a separate piece of hardware called the debug adapter. This device converts signals from the host computer (either program code or debugging commands) into the vendor-specific communication protocol the MCU uses to accept these messages and return its responses. The next five elements are pieces of software which run on the host computer; they can be set up and used separately or are often combined by a number of vendors and third-party companies into an integrated development environment (IDE). A code editor is used by the developer to write application code, and this is sometimes combined with a software development kit (SDK), which simplifies the process of configuring the MCU and interfacing with its various peripherals. SDKs, while not strictly necessary, can greatly reduce the amount of time it takes to turn a functional description of what an MCU needs to do into executable code. Once the application code is written, it must be compiled and linked into an executable in the format needed by the MCU. The adapter driver converts program code and debugging commands from either the IDE, the command line, or a piece of debug software into the vendor-specific communication protocol the debug adapter uses to accept and return these messages. (For the curious, who might ask why the debug adapter is necessary if there is already a piece of software that is converting program code and debugging commands into a vendor-specific format, the answer lies in the fact that although the adapter driver produces vendor-specific *software* messages, it must still communicate through the host computer's USB or other communications port and so a debug adapter is needed to convert the *hardware* signals back and forth, from USB signals to the MCU's programming/debugging pins.) The process of loading a program onto an MCU using its dedicated programming and debugging interface is also called in-circuit programming (ICP). Lastly, the debug software connects through the adapter driver and debug adapter to the debugging features on the MCU, providing the developer with a number of useful tools to gain visibility into their running code.

Using in-circuit programming, an MCU may be programmed with a piece of firmware called a bootloader, which allows the MCU to be programmed over a standard communications port (e.g. USB, ethernet, serial, etc) without the need for a debug adapter. Programming an MCU using a bootloader is called in-application programming (IAP). The toolchain for IAP is shown in the figure below. Some MCUs can be ordered with a bootloader already programmed into the flash memory, but since this isn't an option that is readily available for the majority of MCUs we will be considering in this paper, since programming the bootloader to the MCU requires a debug adapter anyway, and since the bootloader cannot support standard debugging features, we'll not consider this a viable option for most hobbyists and students.



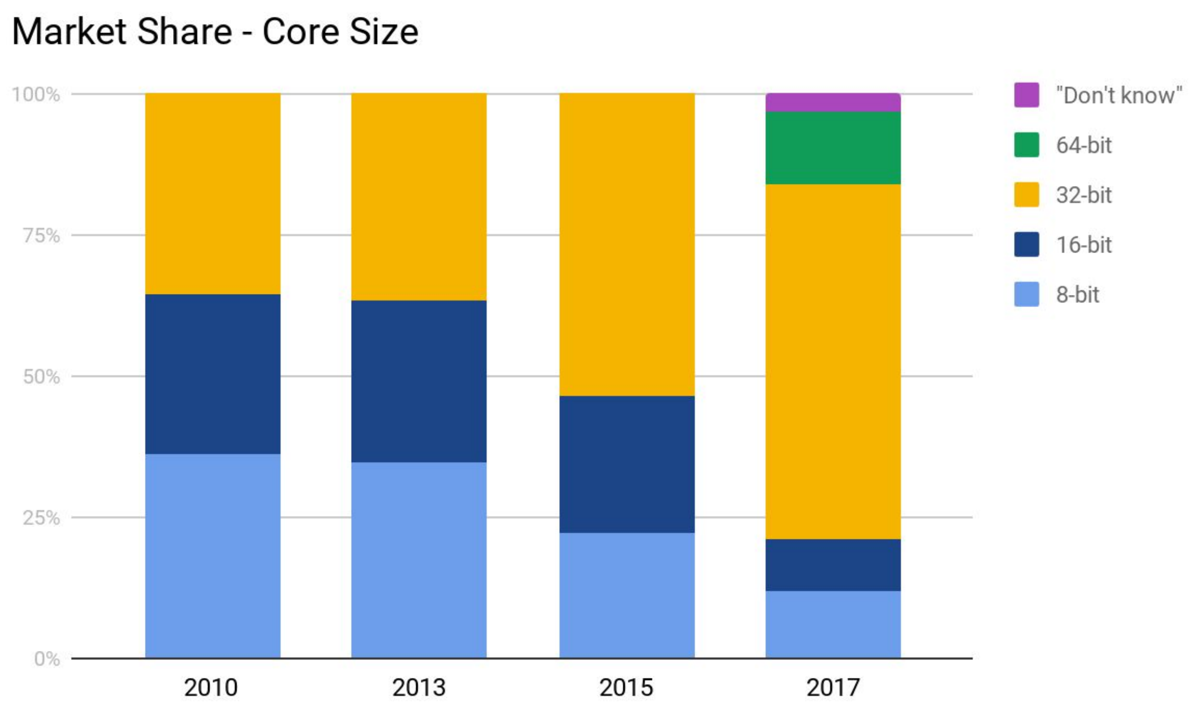
Industry Relevance

The idea of the "industry relevance" of an embedded systems development toolchain is a critical one for this paper and it is defined here to be the relative frequency that each component is used in a commercial design. Unfortunately, there is a dearth of freely available information that would be needed to perform that calculation. (Indeed, although some data was found regarding the industry relevance of embedded microcontrollers, none was found regarding the industry relevance of any other part of the toolchain mentioned previously.) Thus, with the little information available only broad conclusions could be drawn. The data consists of two studies from 2010[8] and 2013[9], both conducted by Frost and Sullivan, a business consultant, and one study called "The McClean Report" conducted by IC Insights, a semiconductor market research company, from the years 2015[10] and 2016[11]. These studies reveal, at a minimum, which semiconductor manufacturers owned the largest portions of the global microcontroller market (the second and third studies state this explicitly and it can also be surmised from the first) and how the market was distributed between the three most common core sizes (8-, 16-, and 32-bit). Only one of the studies reveals which manufacturers owned the largest portions of the global 8-, 16-, and 32-bit microcontroller markets. The last piece of data comes from a survey by AspenCore[12] in 2017 of working professionals in the embedded systems industry. This survey is fairly extensive, and although it cannot be construed as a true market analysis, much can be

[8] "Strategic Analysis of the Global Microcontrollers Market."
[9] "Analysis of the Global Microcontroller Market."
[10] "The McClean Report Contents and Summaries."
[11] "NXP Acquires Freescale, Becomes Top MCU Supplier in 2016."
[12] "ASPENCORE 2017 Embedded Markets Study."

gleaned about the relative popularity of the microcontroller manufacturers and their respective product lines from it.

The most significant conclusion to be drawn relates to microcontroller core size, which was discussed in each study and in the AspenCore survey. This data, shown below, demonstrates the continual rise in popularity of the 32-bit microcontroller and there is little reason to think that this trend has stopped or will stop anytime soon; although 8- and 16-bit microcontrollers maintain some market share due to having sufficient performance for a variety of simple applications at a lower power and lower cost than a 32-bit microcontroller[13], as microcontrollers, overall, get cheaper and better (read: less power hungry) the markets for 8- and 16-bit microcontrollers will continue to dwindle. Given the consistency of the data and the large amount of the microcontroller market which was attributed to 32-bit cores, it can be said that 32-bit microcontrollers are the most industry relevant core size, possibly 2-7 times as relevant as either 8- or 16-bit cores.

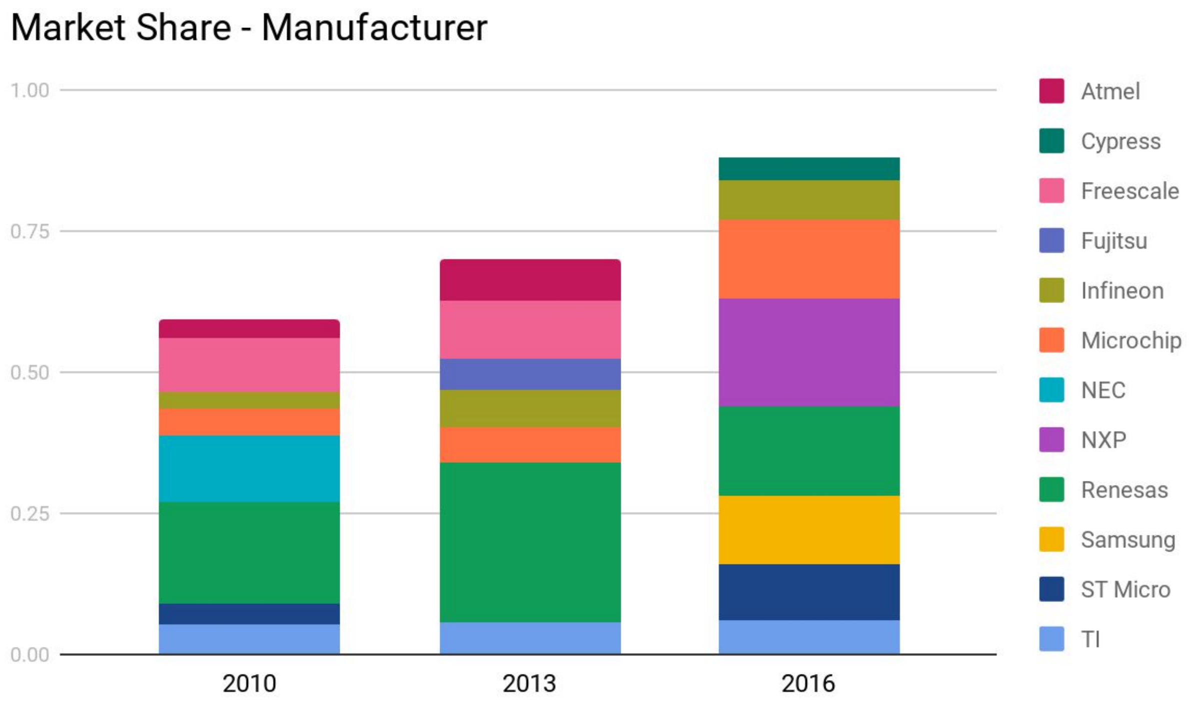


Core size	2010	2013	2015	2017
8-bit	36.1%	34.7%	22.3%	12%
16-bit	28.5%	28.7%	24.0%	9%
32-bit	35.4%	36.6%	53.7%	63%
64-bit	-----	-----	-----	13%

[13] "Analysis of the Global Microcontroller Market."

"Don't know"	-----	-----	-----	3%

The three market studies also included data regarding the top microcontroller manufacturers for the years 2010, 2013, and 2016. Data from the 2010 Frost & Sullivan study was inferred, based on data in that report of the market distribution according to core size (8-, 16-, or 32-bit) and on the leading manufacturers within each core size subcategory (i.e. If company X owned A% of the 8-bit market, B% of the 16-bit market, and C% of the 32-bit market, and if the global market that year was composed of D% 8-bit microcontrollers, E% 16-bit microcontrollers, and F% 32-bit microcontrollers, then company X likely owned (A*D + B*E + C*F)% of the global market in that year). Of note is that NXP merged with Freescale in 2015[14] and Microchip acquired Atmel in 2016[15], which partially explains their absence from the list in 2016 and also the rise in market share of the two acquiring companies. These results show that not only was the majority of the microcontroller market controlled by only about eight of the same companies (NXP/Freescale, Renesas, Microchip/Atmel, ST Microelectronics, Texas Instruments, and Infineon) during the years of study, but also that the percentage of the market controlled by those handful of companies grew between each study; in 2016, the largest eight microcontroller manufacturers owned nearly 90% of the global market.



2010	2013	2016

[14] "Press Release: NXP and Freescale Announce Completion of Merger."
[15] "News Release: Microchip Technology Completes Atmel Acquisition and Provides Update on Its Fiscal Fourth Quarter 2016."

Renesas	17.87%	Renesas	28.5%	NXP	19%
NEC	11.82%	Freescale	10.1%	Renesas	16%
Freescale	9.67%	Atmel	7.4%	Microchip	14%
TI	5.30%	Microchip	6.4%	Samsung	12%
Microchip	4.77%	Infineon	6.6%	ST Micro	10%
ST Micro	3.75%	TI	5.5%	Infineon	7%
Atmel	3.29%	Fujitsu	5.5%	TI	6%
Infineon	2.96%	-----	-----	Cypress	4%

Given the consistency of the data and the large amount of the microcontroller market which was owned by these manufacturers, microcontrollers produced by one of the eight major manufacturers listed above can be said to be distinctly industry relevant. Others possibly are, as well, but none more so than these. It is tempting to rank the industry relevance of each manufacturer based on their market share, but with the age of the data used and the volatility of the individual rankings, the results would be dubious, at best.

Although it would tempting at this point to conflate the two assessments above (by saying, for instance, that 32-bit microcontrollers from NXP/Freescale are the “most” industry relevant microcontrollers and 16-bit microcontrollers from Cypress Semiconductor are the “least” industry relevant), we cannot make this determination with the information present. At any rate, what is of greater interest than which manufacturer's core size held the greatest market share is the more specific question of which manufacturer’s various product lines held the greatest market share. This information was absent from each of the data sources, but could be roughly estimated using the AspenCore survey. This method is based on two questions from the survey, which were “My current embedded project's main processor is a [8/16/32/64/Don’t know]-bit processor” and “Which of the following 32-/16-/8-bit chip families would you consider for your next embedded project?”. Assuming there is some correlation between the core sizes that were then being used and the product lines which were being considered for use in the respondents’ next embedded project, we can produce the table below. AspenCore survey respondents were allowed multiple answers to the second question, which is why their responses add up to more than 100%.

Manufacturer	Core size (bits)	Product line	Percent responses
ST Micro	32	STM32	19%
Microchip	32	PIC32	13%

Xilinx	32	Zinq (w/ dual M9)	11%
NXP (Freescale)	32	i.MX	11%
NXP	32	LPC	10%
NXP (Freescale)	32	Kinetis	10%
Microchip (Atmel)	32	SAMxx	9%
Texas Inst.	32	Sitara	9%
Intel	32	Atom, Pentium, Celeron, Core 2, Core IX	8%
Intel	32	Altera	8%
Microchip (Atmel)	32	Arduino	8%
Texas Inst.	32	Simplelink	7%
Texas Inst.	32	TM4Cx	7%
Microchip (Atmel)	32	AVR32	7%
Microchip (Atmel)	32	AT91/SAM	6%
Cypress	32	PSoC4/5	6%
Microchip	8	PIC10/12/16/18	6%
Microchip (Atmel)	8	AVR	5%
Renesas	32	RX	5%
Broadcom	32	Any	5%
Texas Inst.	32	C2000	4%
Microchip	16	PIC24, dsPIC	4%
Texas Inst.	16	MSP430	4%
ST Micro	8	STM6/7/8	2%
ST Micro	16	STM9/10	2%

The manufacturers listed above mostly match those listed previously as having the largest market share, albeit in a slightly different order. Also included on this list, which were absent from the market studies, were systems-on-chips (SoCs; ex: Xilinx Zynq, Intel Atom/Celeron/etc, Broadcom) and field-programmable gate arrays (FPGAs; ex: Intel Altera). Given the timeliness of the data and the manner in which it agrees with the previously discussed market studies, it can be said with moderate confidence that any microcontroller listed on the table above is industry relevant. It is tempting, as before, to rank the industry relevance of each product line based on their response percentages, but without further information about the underlying data, the results would be dubious, at best.

Regarding toolchains, in spite of the lack of data regarding their industry relevance, there are two features which will here be required to consider a toolchain as "industry relevant"[16]:

- Source-level debugging: the ability to set source-level breakpoints and to step through code one line at a time.
- The ability to manipulate registers and to directly implement assembly code. In other words, development is not constrained by any OS or framework other than that intentionally included by the developer.

Although it is not unheard of for commercial developers to work with toolchains that lack one or both of these features, they are typically required for any project of any appreciable complexity and efficiently describe the difference between toolchains designed primarily for industry and those designed primarily for education.

Challenges faced by hobbyists, students, and educators

As in many other professional disciplines, producing high-quality embedded systems is difficult, even for seasoned developers at relatively large companies. Consider the depth and breadth of knowledge required of embedded systems developers, as described by these researchers:

- "The co-existence of various kinds of devices, protocols, architectures, and applications make internet of Things (IoT) systems complex to develop, even for experienced programmers."[17]
- "Historically, embedded systems development requires knowledge of low-level programming languages, local installation of compilation toolchains, device drivers, and applications. For students and educators, these requirements can introduce insurmountable barriers."[18]
- "This [developing a working knowledge of microcontrollers] requires both an understanding of all the ancillary system elements, the capabilities of the microcontroller, and system integration techniques—no mean feat!"[19]

For hobbyists, students, and educators (who frequently lack the experience, time, money, and/or resources that large companies enjoy), these barriers can easily place industry-relevant embedded systems development out of their reach. To better understand this environment, we

[16] Staver et al., "Encouraging Interest In Engineering Through Embedded System Design"; Mondragon and Becker-Gomez, "So Many Educational Microcontroller Platforms, So Little Time!"
[17] Corno, De Russis, and Saenz, "On the Challenges Novice Programmers Experience in Developing IoT Systems: A Survey."
[18] Devine et al., "MakeCode and CODAL: Intuitive and Efficient Embedded Systems Programming for Education."
[19] Barnes et al., "Teaching Embedded Microprocessor Systems by Enquiry-Based Group Learning."

will use work done by Dr. Joel Sadler, et al[20]; Fulvio Corno, et al[21]; James Devine, et al[22]; Essi Lahtinen, et al[23]; and others, to enumerate 10 of these challenges. We will organize these challenges into three groups, corresponding to the chronology of when the hobbyist or student might use them: “Assembling the necessary tools and components”, “Developing the embedded system”, and “Debugging”. “Developing the embedded system” encompasses the activities of writing firmware, designing circuits, and building prototypes. “Assembling the necessary tools and components” involves anything that must happen prior to the activity of developing and “Debugging” involves any activity after the initial development effort to get the embedded system working per the design specifications.

- Assembling the necessary tools and components
    - Cost and availability of tools and components: At a minimum, the student needs to acquire the MCU with any necessary supporting hardware (such as power filtering capacitors, external oscillators, power sources, etc) and a debug adapter. Although individual MCUs can often be found quite cheap, even in single quantities, debug adapters can cost into the hundreds or thousands of dollars, with only a few costing less than $50. As the required tools and components are most often only available through an online retailer, they frequently have a lead time of one to four weeks. They can often be purchased as a single unit, called a development or breakout board, alleviating the student of needing to design the circuit themselves. These boards frequently cost between $10 and $20, though a few can be had for $6 or less. However, the cost of these boards typically exceeds that of an individual MCU and one of the less expensive debug adapters after only a handful of units, making them mostly unsuitable for use in quantity. If the circuit is to be designed and built in-house, which may also be required if any other part of the circuit in the embedded system to which the MCU is to connected cannot be easily prototyped on a standard breadboard, the student must also then acquire the requisite fabrication tools and materials (also frequently having a one to four week lead time) as well as possess the skills required to confidently construct the necessary circuit, complete the PCB design, fabricate the PCB, and populate the PCB. This is roughly equivalent to an undergraduate-level knowledge of electrical engineering.
    - Software download, installations, and configuration/setup: At a minimum, the student must prepare a development environment with four of the five components mentioned previously (code editor, compiler, adapter driver, and debug software). Although many MCU manufacturers will prominently display

[20] Sadler, Shluzas, and Blikstein, “Abracadabra: Imagining Access to Creative Computing Tools for Everyone”; Analytis, Sadler, and Cutkosky, “Creating Paper Robots Increases Designers’ Confidence to Prototype with Microcontrollers and Electronics”; Sadler, Shluzas, and Blikstein, “Building Blocks in Creative Computing: Modularity Increases the Probability of Prototyping Novel Ideas.”
[21] Corno, De Russis, and Saenz, “On the Challenges Novice Programmers Experience in Developing IoT Systems: A Survey.”
[22] Devine et al., “MakeCode and CODAL: Intuitive and Efficient Embedded Systems Programming for Education.”
[23] Lahtinen, Ala-Mutka, and Jarvinen, “A Study of the Difficulties of Novice Programmers.”

    their internal IDE, which is almost always guaranteed to work for any of their MCUs, downloading and installing such programs is a non-trivial task, especially since many of these programs have unique software dependencies that aren't maintained by the MCU manufacturer. Additionally, the student must ensure that the software is configured correctly for use with their MCU, which often requires navigating a series of buttons and options inside the IDE that assume the user has a certain level of professional experience that many students and hobbyists do not have.
    - Lack of feedback regarding assembling tools and components: Aside from obvious errors, such as mis-matched connectors or software error messages, there is little offered to the student in the way of feedback that might indicate to them if they've assembled the tools and components above correctly. Some development software will provide a feature to test a connection with the target MCU and this can be very useful to ensure that the above steps have been completed correctly, but many do not. For those pieces of software, the only way to test if the MCU and software are configured correctly is to write a simple program and attempt to program the MCU. Of course, if this doesn't work then the student is left with any number of reasons why the programming failed, from a faulty MCU power source, to an incorrect connection between the MCU and debug adapter, to an incorrect software setting.
- Developing the embedded system
    - Programming the MCU: Aside from knowing how to write correct code, the student must also understand how to configure the underlying hardware to do what they want. Although many MCU manufacturers will provide some form of software development kit (SDK) to make this process easier, using such without understanding what it is doing is a recipe for subtle and hard-to-identify bugs that could derail an embedded project. In most cases, there is a distinct lack of support documentation that is readily understood by the novice. Instead, and in fantastic irony, most MCUs are accompanied by datasheets and manuals frequently totaling over one thousand pages of dense, technical text. Fully understanding how the hardware works essentially requires an undergraduate-level knowledge of digital design and computer engineering, and the confidence and patience to wade through hundreds of pages of technical documentation.
    - Usability of the tools: Although the design of graphical user interfaces (GUIs) has greatly improved the usability of the software tools used to program an MCU, there still exists a (mostly) mutually exclusive relationship between the software's usability and the features it offers. Unfortunately, most development software prioritizes features (those most requested from developers in industry, it is presumed) over usability. This often manifests as crowded displays (too many windows or "perspectives"), crowded toolbars, deeply nested menu options, and the requirement that a user configure multiple settings in order to perform basic tasks (e.g. selecting a compiler toolchain before starting a new project).

    - Cost and availability of prototyping tools and materials: To consider development complete, the MCU and circuit must often be moved from a temporary prototyping area (such as a breadboard) to a permanent installation (such as a protoboard or custom PCB). Connecting to all of the input and output devices (including power) also typically requires several wire-to-board connections and often that the student hand-makes those wires. The entire embedded system is also typically mounted inside an enclosure. Any of the tools and materials required to do this likely have a one to four week lead time. Many tools (such as wire crimpers or 3D printers) can be quite expensive.
    - Breadth of domain knowledge required: Developing embedded systems requires domain knowledge in several different highly technical fields, such as electrical engineering, software engineering/computer programming, and computer engineering (as was mentioned in the opening paragraph to this section). One researcher noted that “an embedded design engineer must also be comfortable with the concepts of thermal loading, the MIPS/watts ratio, an intimate knowledge of the target hardware to the register level, and the software interaction with the registers. Furthermore, the engineer must be aware of the overhead involved with the specific choice of software compiler employed to program the target system [1].”[24]
    - Application-specific design considerations: The student must design their embedded system so that not only is it functional on the lab bench, but also that it works under all desired conditions and constraints. This includes such things as ensuring the enclosure is weatherproof if the embedded system is to be installed or used outside or incorporating security considerations if the embedded system is to have wireless connectivity or be of value to criminals.
- Debugging
    - Lack of visibility into the MCU: Microcontrollers, even simple ones, are vastly complex instruments whose outputs depend on a host of state variables (such as the current program counter, the status of peripheral registers, the status of the memory buses, which instructions are in the pipeline [if one is implemented], etc). Of course, it’s not possible to physically probe an internal node of an MCU nor is it possible to connect any internal buses to a logic analyzer. With a debug adapter and the proper software tools, developers can get rather close to this ideal (being able to inspect individual registers and control program execution, more or less), but the fact remains that doing so is like trying to view the world through a soda straw.
    - Multiple modes of failure: The inherent complexity of a mostly or fully completed embedded system means that error symptoms have a vast number of possible root causes, from component malfunction to failure of an electrical connection to software bugs to incorrectly configuring the MCU for operation. These root causes get harder to identify the more they’ve been abstracted by the developer

[24] Barrett et al., “Embedded Systems Design: Responding to the Challenge.”

or development toolchain, which is a matter of course for embedded development. Things like incorrect programming code, component failure, or electrical connection failure are relatively easier to diagnose because they are “staring the developer in the face”, so-to-speak. If the root case is a stack overflow or improperly configured interrupt register, however, then these are harder to diagnose because they are not often explicitly manipulated by the developer or aren’t conveniently inspected during normal operation.

To put this list another way, there are myriad ways for an embedded system to fail and this, ultimately, is what makes development difficult. In an experiment in which participants were asked to control LEDs from a microcontroller, Dr. Joel Sadler found no less than fifteen different categories describing the ways that participants failed or encountered barriers, ranging from “Software usability error” to “LED polarity error”[25]. And as any teacher or game developer will tell you, when students or players are faced with challenges that are too great for their abilities, they will lose interest and find something else to do. For industry, this is a relatively low concern but for hobbyists and students, especially those mentioned previously as being underrepresented in STEM or who lack access to STEM education, this sometimes makes the difference between a career in STEM and a different one entirely. To improve accessibility to industry-relevant embedded systems development, these challenges must be addressed.

Problem Statement

Identify the tools and materials which are best suited for hobbyists, students, and educators to learn about and prototype embedded systems; evaluation criteria should prioritize (in no particular order) industry-relevance, ease of use, and low cost. Where deficiencies are identified, develop novel tools and materials to both increase the number of development tools which are accessible to this demographic and to improve the development tools’ ease of use and/or cost.

Finding Hobbyist-friendly, Industry-relevant Microcontrollers

Search methodology

The search for which MCUs are available to the hobbyist, student, and educator began with component distributors who can provide some assurance that the parts they sell are genuine (for more information about the problem of counterfeit electronic components, see Appendix A: The Problem of Counterfeit Electronic Components). The author primarily searched for components on Digi-key Electronics[26], although also included Mouser Electronics[27], Future

[25] Sadler, Shluzas, and Blikstein, “Building Blocks in Creative Computing: Modularity Increases the Probability of Prototyping Novel Ideas.”
[26] https://www.digikey.com/
[27] https://www.mouser.com/

Electronics[28], and Arrow Electronics[29] in their searches. The author also used two aggregate websites, ECIA Authorized[30] and Source Engine[31], to search for the availability of specific parts.

Each distributor has organized their inventory by category, with one such category being "microcontrollers" (though the actual name varies by distributor). Each distributor's inventory was narrowed to this category and then filtered to include only those parts which were in-stock, had an "active" status (as opposed to "obsolete" or "not for new designs"), and were available in single quantities. This was to ensure that the MCUs being considered were readily available (in-stock), would likely be so for some time to come ("active" status), and did not require bulk purchase (for hobbyists or students who might only need a single MCU for a class or one-off project).

At this point, it was determined that "ease of use" would most significantly segregate the incredibly long list of MCUs which met the author's search criteria so far. For hobbyists, students, and educators, it was assumed that access to printed circuit board (PCB) fabrication and assembly would be limited and/or costly, so the MCUs which would be most easily prototyped with were those that came in a dual in-line package (DIP) or those that came on a breakout or development board (in that order). Here, we are assuming that though the developers might not have access to PCB fabrication and assembly, they would likely have access to breadboards and to prototyping boards with 100 mil pin spacing (e.g. stripboards, protoboards, etc); both are suitable bases on which to prototype either a DIP MCU or a breakout board. Next are MCUs that come in a small outline integrated circuit (SOIC) or plastic leaded chip carrier (PLCC) package; both packages with 50 mil pin spacing. Partly based on the author's experience, it was determined that for developers who can comfortably solder through-hole components (i.e. those with 100 mil pin spacing like the DIP MCUs mentioned above), soldering components with 50 mil pin spacing would pose a small but not significant challenge. With the little bit of additional work required to procure an SOIC or PLCC breakout board, a hobbyist or student could conceivably prototype with any MCU that came in that package. Lastly, the author considered MCUs in all other packages. This last tier represents the MCUs that are the most difficult to work with, as their reduced pin spacing makes assembly quite difficult and would require either advanced surface-mount soldering skills or external PCB fabrication and assembly.

After sorting the list of MCUs by the four levels of difficulty listed above (DIP MCUs, MCUs on breakout boards, SOIC/PLCC MCUs, and all other MCUs), the results were then sorted by price from low to high and evaluated. Breakout boards costing more than $8 per unit and MCUs in an SOIC/PLCC/SDIP package costing more than $2 were excluded from consideration because the ratio of MCU performance to their cost fell steeply beyond that threshold. Importantly, the $2 threshold for SOIC/PLCC/SDIP MCUs did not include any components necessary to program the MCU and confirm that the programming was successful, which typically included the following components:

- PCB;

[28] https://www.futureelectronics.com/
[29] https://www.arrow.com/
[30] https://www.eciaauthorized.com/en
[31] https://www.sourcengine.com/

- programming pins (typically male pin headers);
- a power filtering capacitor;
- a power LED and current limiting resistor;
- a user LED and current limiting resistor; and
- a reset sub-circuit (typically pull-up resistor, capacitor, and pushbutton).

In that a rough estimate for these components is $1.50 ($0.50 for the PCB and $0.10 for every other component), this meant that the total cost to prototype with an SOIC/PLCC/SDIP MCU became, roughly, $3.50 (a value just slightly lower than the cheapest development boards I could find).

As most development software for these MCUs is free, the debug adapter represents the next and final purchase a developer needs to make. Unfortunately, debug adapters can be quite expensive (as much as a thousand dollars[32]) but some models can be had for much less money; the author found the following, each costing less than $25:

- Segger J-Link EDU Mini[33]
- Microchip MPLAB SNAP[34]
- Texas Instruments ez-FET on a Launchpad development board[35]
- ST-LinkV2 (reduced) from various third-parties[36]
- ST Microelectronics ST-LinkV2[37]
- ST Microelectronics ST-LinkV3 Mini[38]
- Nuvoton Nu-Link[39]
- CMSIS-DAP[40]
- FTDI USB-to-UART and USB-to-Multiprotocol adapters[41]

Debug adapters costing more than about $25 were excluded from consideration for two reasons: first, because the next most expensive debug adapters cost around $50 and an approximate $25 limit established a natural breakpoint; and second, because the remaining debug adapters that cost less than about $25 were still compatible with a large section of the MCU market. Thus, the results from above were filtered manually to include only those MCUs which were compatible with the debug adapters listed above.

[32] Segger J-Link PRO from Digi-Key (https://tinyurl.com/y75xb2v5)
[33] Segger J-Link EDU Mini from Digi-key (https://tinyurl.com/y8v7cs34)
[34] Microchip MPLAB SNAP from Digi-key (https://tinyurl.com/ya9taszl)
[35] TI MSP-EXP430430G2ET development board (https://tinyurl.com/y9bchbyx)
[36] ST-LinkV2 from Seeed Studios (https://tinyurl.com/ycyqv2g5)
[37] ST Microelectronics ST-LinkV2 (https://tinyurl.com/kkerdrm)
[38] ST Microelectronics ST-LinkV3 (https://tinyurl.com/y8at9kd4)
[39] Nuvoton Nu-Link (https://tinyurl.com/y8s882ep)
[40] L-Tek SWDAP Interface (https://tinyurl.com/y9f6oh3j); also available on most NXP development boards, such as the FREDM-KL25Z (https://tinyurl.com/y9fvggoe)
[41] Only for use with Black Magic Probe or OpenOCD. Not all FTDI boards will work with these two adapter drivers, so you'll need to either make sure it's listed as a piece of supported hardware or experiment with it once you get it to see if it will work.

MCU List

DIP MCUs

Only three families of MCUs were found in a DIP package that met the criteria established above: the PIC16/PIC18 (Microchip), the dsPIC33 (Microchip), and the MSP430 (Texas Instruments). Not every PIC16/PIC18/dsPIC33 met the criteria, however, since not every PIC16/PIC18/dPIC33 is compatible with the MPLAB SNAP (Microchip's only low-cost debug adapter). Finding which ones are requires either navigating to each MCUs' product page and seeing if the MPLAB SNAP is listed under supporting debuggers or downloading the latest version of MPLAB X IDE and opening a file called "DeviceSupport.htm", which contains a matrix of which MCUs are supported by which debug adapter.

Manufacturer	**MCU**	**Core**	**Cost**
Microchip	PIC	PIC	$0.80+[42] (Digi-Key)
Texas Instruments	MSP430	MSP430	$1.34+[43] (Digi-Key)

Breakout Boards

Manufacturer	**MCU**	**Core**	**Cost**
Espressif	ESP8266	Tensilica L106	~$4.00[44] (Olimex)
Microchip	ATSAMD21	Cortex-M0+	$4.90[45] (Seeed St.)
Seeed Studios	Sipeed Longan Nano	RISC-V	$4.90[46] (Seeed St.)
Seeed Studios	Sipeed Tang Nano	FPGA	$4.90[47] (Seeed St.)
Microchip	ATSAMD09	Cortex-M0+	$4.95[48] (Adafruit)
NXP	LPC845-BRK	Cortex-M0+	$5.98[49] (Digi-Key)
Infineon	XMC2GO	Cortex-M0	$6.12[50] (Digi-Key)

[42] https://tinyurl.com/ybtl2cwz
[43] https://tinyurl.com/ydyj2mlf
[44] https://tinyurl.com/ybujtqll
[45] https://tinyurl.com/sf5hof6
[46] https://tinyurl.com/y5e2cpcv
[47] https://tinyurl.com/qoltnc7
[48] https://www.adafruit.com/product/3657
[49] https://www.digikey.com/products/en?keywords=lpc845-brk
[50] https://tinyurl.com/ycwwu5om

Silicon Labs	EFM8BB1LCK	8051	$6.65[51] (Digi-Key)
Seeed Studios	Seeed GD32	RISC-V	$6.90[52] (Seeed St.)
ST Microelectronics	STM8SV-DISCO	STM8	$7.00[53] (Digi-Key)
ST Microelectronics	STM32L100-DISCO	Cortex-M3	$7.88[54] (Digi-Key)

SOIC/PLCC/SDIP MCUs

Manufacturer	**MCU**	**Core**	**Cost**
Cypress	PSoC4	Cortex-M0	$0.19+[55] (Digi-Key)
Silicon Labs	EFM8/C8051	8051	$0.59+[56] (Digi-Key)
ST Microelectronics	STM8	STM8	$0.63+[57] (Digi-Key)
Microchip	PIC	PIC	$0.64+[58] (Digi-Key)
Microchip	ATSAMD09/10/11	Cortex-M0+	$0.94+[59] (Digi-Key)
ST Microelectronics	STM32G0	Cortex-M3	$0.94[60] & $1.43[61] (Digi-Key)
Texas Instruments	MSP430	MSP430	$1.05+[62] (Digi-Key)
NXP	LPC812	Cortex-M0+	$1.58[63] (Digi-Key)
NXP	KE04	Cortex-M0+	$1.83[64] (Digi-Key)

All Other Device Packages

Being able to hand-solder smaller packages than SOIC/PLCC allows the hobbyist to prototype with many more MCUs than those listed above. Those lacking in that skill will require the use of a PCB assembly service to work with these MCUs. This is a powerful way for

[51] https://www.digikey.com/product-detail/en/silicon-labs/EFM8BB1LCK/336-6118-ND/10294225
[52] https://www.seeedstudio.com/SeeedStudio-GD32-RISC-V-Dev-Board-p-4302.html
[53] https://www.digikey.com/products/en?keywords=STM8SV
[54] https://tinyurl.com/yaxee8on
[55] https://tinyurl.com/yczwuj3b
[56] https://tinyurl.com/y9qfhvxr
[57] https://tinyurl.com/y98km94l
[58] https://tinyurl.com/yca54pqb
[59] https://tinyurl.com/ycwwfja5
[60] https://tinyurl.com/ybzfqmje
[61] https://tinyurl.com/y998xccb
[62] https://tinyurl.com/y8xwbwkm
[63] https://tinyurl.com/yc2h5x2n
[64] https://tinyurl.com/yb8m6fkb

hobbyists, students, teachers, and makers to get access to surface-mount components that they themselves might not have been able to assemble. Below are two such PCBs that the author designed so as to be assembled by JLCPCB. At $3 and $3.30, they are some of the cheaper options for "development boards" that have yet been identified.

Manufacturer	**MCU**	**Core**	**Cost**
ST Microelectronics	STM32F0	Cortex-M0	~$3.00[65] (JLCPCB)
ST Microelectronics	STM32G0	Cortex-M3	~$3.30[66] (JLCPCB)

Evaluating the Associated Toolchains

Six of the above MCUs, the ones roughly corresponding to the cheapest and easiest to work with, were then procured and programmed using a total of 11 different IDEs and five different debug adapters. This formed the preponderance of the work for this independent study. The six MCUs were:

- the PIC16F18446,
- the dsPIC33EV32GM002,
- the MSP430G2553,
- the XMC2GO development board (using the XMC1100Q024F0064),
- the LPC845-BRK development board (using the LPC845M301JBD48), and
- a custom-built development board for the STM32G031J6M6.

How easy each was to work with was predominantly a function of the development environment and the documentation and support that was available.

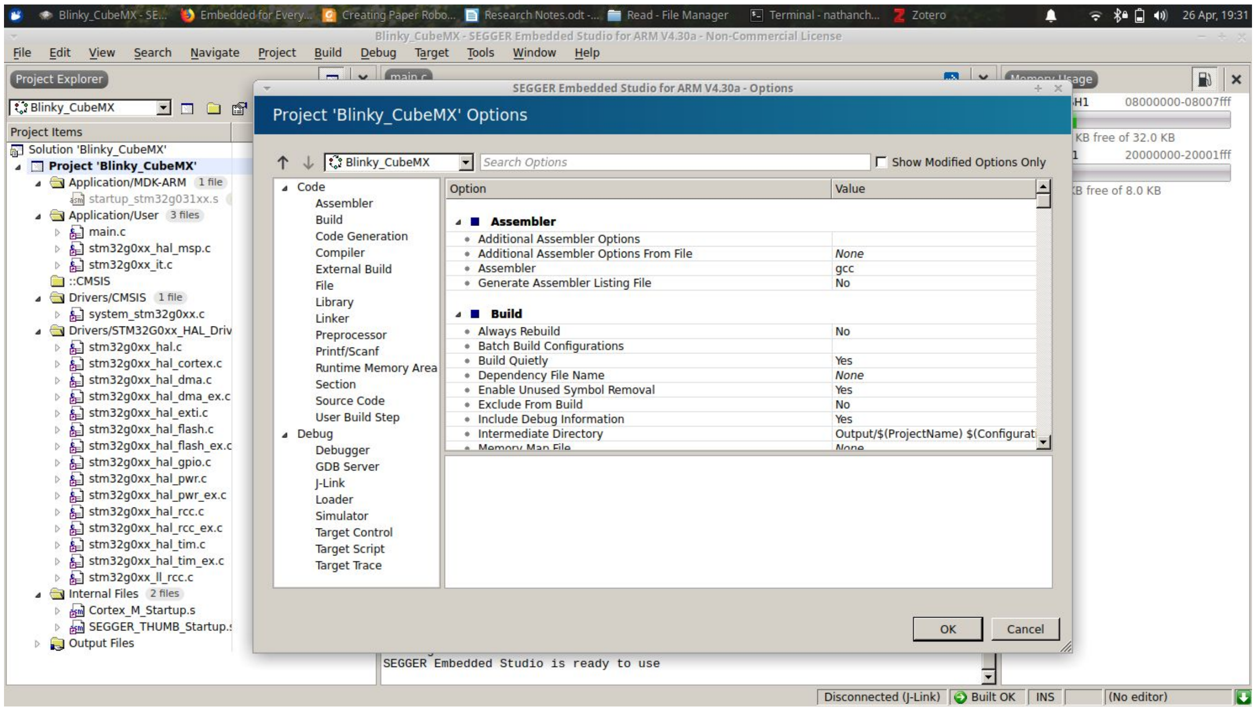
Development Environments

Development environments for the six MCUs fell into roughly three categories: vendor-supplied, third-party, and command-line. Only one online IDE was available (CCS Cloud for the MSP430). All were based on graphical user interfaces except the command-line (clearly), and they varied widely in their ease of use based on two factors:

- how easy or difficult it was to install the IDE, and
- how many features the IDE included.

Many tools required external dependencies that were not included into the software download alongside the IDE and some installations seemed more brittle or error-prone than others. Once installed, most IDEs opened a “Welcome” or “Start” page to help guide the new user appropriately but also a bewildering array of menus and options (see the image below, taken from one of the author’s projects using Segger Embedded Studio). Users are allowed to configure dozens of settings for the overall project, for individual project files, and for various debug and release versions.

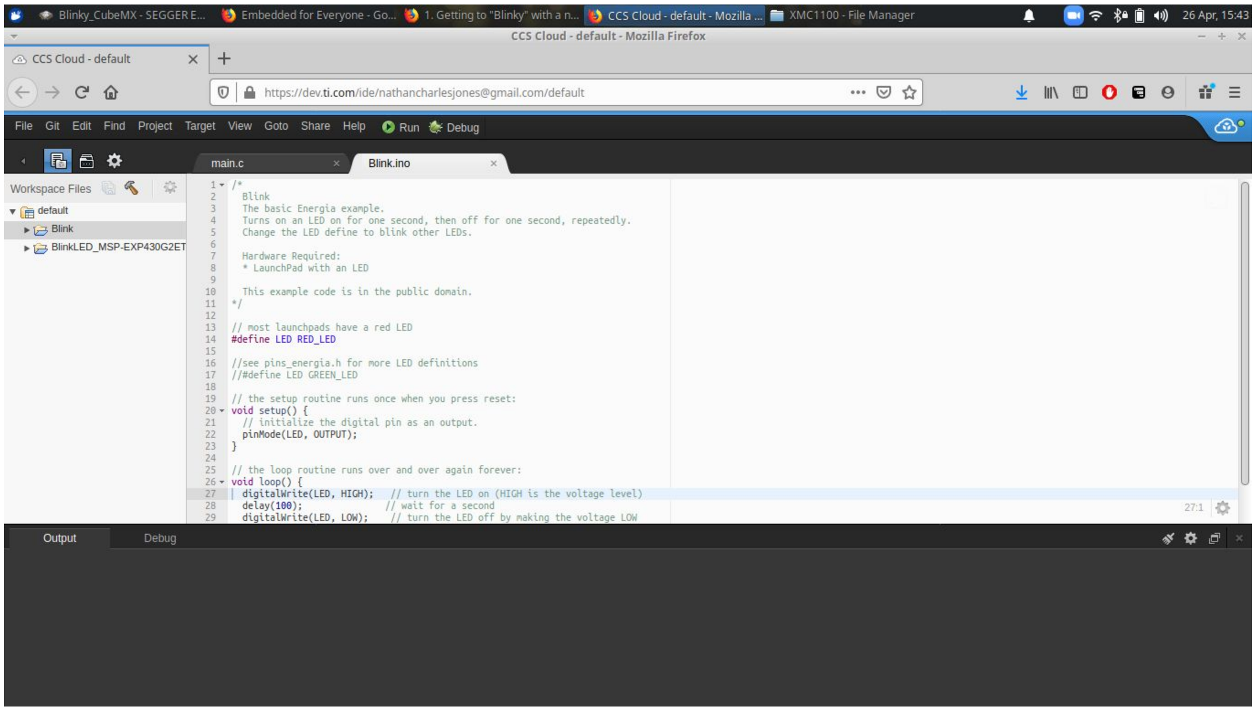
[65] https://github.com/nathancharlesjones/STM32F030F4P6-breakout-board.git
[66] https://github.com/nathancharlesjones/STM32G0-Breakout-Boards.git



Option	Value
Assembler	
Additional Assembler Options	
Additional Assembler Options From File	*None*
Assembler	gcc
Generate Assembler Listing File	No
Build	
Always Rebuild	No
Batch Build Configurations	
Build Quietly	Yes
Dependency File Name	*None*
Enable Unused Symbol Removal	Yes
Exclude From Build	No
Include Debug Information	Yes
Intermediate Directory	Output/$(ProjectName) $(Configurat
Memory Map File	*None*

```
SEGGER Embedded Studio is ready to use
```

Many also had a complicated process to start a new project, requiring the user to make such decisions as whether they wanted their program to run from Flash or RAM, whether they were building an application or library code, and even which compiler they wished to use. For hobbyists whose only experience with embedded systems development is the Arduino IDE, these can seem like inscrutably complex options. The simplest IDE to use was CCS Cloud for the MSP430, owing to the fact that it required no local installation beyond a single browser plug-in (it is on online IDE) and also to the fact that it presents a highly simplified interface to the user (see image below; the two main features of this IDE are the “Run” and “Debug” buttons).



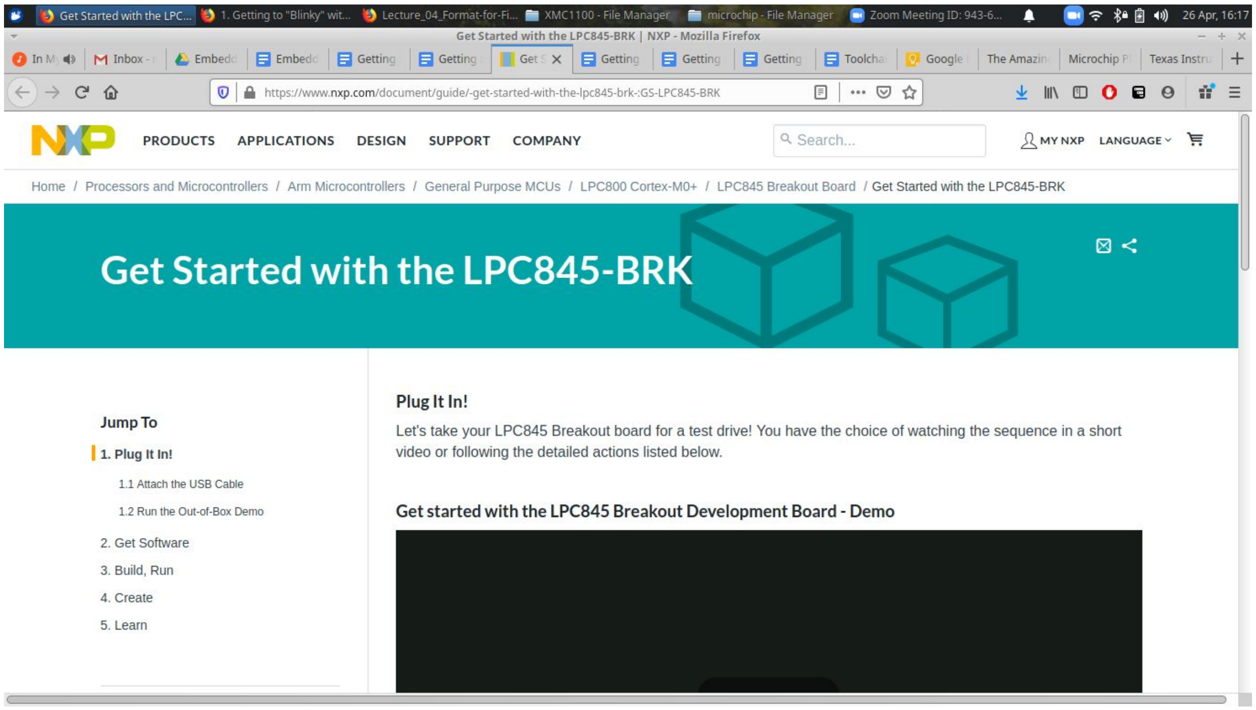
CCS Cloud - default - Mozilla Firefox
https://dev.ti.com/ide/nathancharlesjones@gmail.com/default
File Git Edit Find Project Target View Goto Share Help Run Debug
main.c
Blink.ino
Workspace Files
default
Blink
BlinkLED_MSP-EXP430G2ET
Output
Debug

```
/*
  Blink
  The basic Energia example.
  Turns on an LED on for one second, then off for one second, repeatedly.
  Change the LED define to blink other LEDs.

  Hardware Required:
  * LaunchPad with an LED

  This example code is in the public domain.
*/

// most launchpads have a red LED
#define LED RED_LED

//see pins_energia.h for more LED definitions
//#define LED GREEN_LED

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
  pinMode(LED, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
  digitalWrite(LED, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(100);               // wait for a second
  digitalWrite(LED, LOW);    // turn the LED off by making the voltage LOW
```

Code generation was most robust for the STM32, through a standalone software application called “STM32CubeMX”. In CubeMX, the developer can select whether they want to build a project for a number of different IDEs, including MDK-Arm, IAR, and a makefile-based project. In fact, it was the makefile from CubeMX that the author edited to be able to build projects for the XMC1100 and the LPC845 from the command line much more robustly than he had figured out how to do on his own.

Documentation and Support

Although each MCU had its own product page on the manufacturer’s website, with its associated datasheets, reference manuals, programming manuals, and sell sheets, the nature of this information is such that it does no favors for the individual who is simply trying to “get started” with a new MCU (unless that individual is highly experienced and knows exactly where to look for the information they need). Two notable exceptions were the LPC845 and the STM32. In the case of the LPC845, the manufacturer, NXP, has actually gone to the trouble of creating a “getting started” webpage, with information about how to download the required software, how to build and run a basic example project, how to create new projects, and where to go for additional information. It leaves a bit to be desired, but it’s absolutely a step in the right direction.


Get Started with the LPC845-BRK | NXP - Mozilla Firefox
https://www.nxp.com/document/guide/-get-started-with-the-lpc845-brk-:GS-LPC845-BRK
PRODUCTS
APPLICATIONS
DESIGN
SUPPORT
COMPANY
Search...
MY NXP
LANGUAGE

Home / Processors and Microcontrollers / Arm Microcontrollers / General Purpose MCUs / LPC800 Cortex-M0+ / LPC845 Breakout Board / Get Started with the LPC845-BRK

Get Started with the LPC845-BRK

Jump To

1. Plug It In!
   1.1 Attach the USB Cable
   1.2 Run the Out-of-Box Demo
2. Get Software
3. Build, Run
4. Create
5. Learn

Plug It In!

Let's take your LPC845 Breakout board for a test drive! You have the choice of watching the sequence in a short video or following the detailed actions listed below.

Get started with the LPC845 Breakout Development Board - Demo

The STM32 had no such “getting started” webpage, but it did have a much greater quantity of information on it’s product page than did the other MCUs, which included a number of useful and interesting application notes about how to use the MCU as well as information about which IDEs and SDKs were officially supported by the MCU. Additionally, it should be noted that the STM32 is quite likely the most commonly used MCU in industry and it has strong support in the hobbyist community, which can make a big difference in how easy it is to get started with an MCU.

Navigable and searchable API references were available for the vendor-supplied SDKs for the XMC1100, LPC845, and STM32, and it’s assumed that one could also be found for the other three MCUs had the author looked hard enough. The API reference for the LPC845 was of the highest quality, being easy to find and including critical background information about the API (as opposed to simply descriptions of the functions and defines). The API reference for the STM32 was slightly more difficult to find and navigate, being a hyperlinked PDF document as opposed to a webpage. The API reference for the XMC1100 was the most difficult to find of the three (in fact, the author only found it almost by accident) and it contained the least amount of information.

The “Embedded for Everyone” Wiki

In order to make available the “getting started” guides, makefiles, scripts, example projects, and custom printed circuit boards (PCBs), that were written or created in the course of this independent study, and to sufficiently introduce and explain them, a Git repository[67] was

[67] https://github.com/nathancharlesjones/Embedded-for-Everyone

created with an associated wiki[68]. The wiki was organized as follows, in order to roughly match the chronological order in which a hobbyist or student would need to know them (similar to the ordering above from the section “Challenges faced by hobbyists, students, and educators”):

- Getting to “Blinky” with a new MCU
    - Step 1: Understand the Parts of an Embedded Toolchain
    - Step 2: Pick a New MCU
        - DIP
        - MCU breakout/development boards
        - SOIC/PLCC/SDIP
        - All other device packages
    - Step 3: Find a "Getting Started" Guide
        - Getting Started with the PIC16/PIC18/dsPIC33
        - Getting Started with the MSP430
        - Getting Started with the XMC2GO (XMC1100)
        - Getting Started with the LPC845
        - Getting Started with the STM32G0
    - Recommendations for Moving Beyond an Educational Platform
- Developing application code for (and debugging) an MCU
    - Software development kits (SDKs)
        - Finding Start-up Code and a HAL
        - Finding Middleware and Drivers
        - A Short List of Useful SDKs
        - Using an SDK
    - Software architectures and design patterns
    - General Debugging Tips
    - Debugging Tools and Techniques
        - Multimeter
        - Debug adapter
        - Multi-function Instrument
        - USB-to-UART bridge or Multi-protocol Adapter
- Building a circuit on a PCB and connecting it to the rest of the embedded device
    - Step 1: Find the right parts
        - Sourcing parts and avoiding counterfeits
        - Tips when searching for parts
    - Step 2: Build the circuit on a breadboard
    - Step 3: Lay out the PCB or get a protoboard
        - Custom stripboard and breadboard-style protoboards
    - Step 4: Order the PCB
        - Custom STM32F0 breakout board for ~$3.00 from JLCPCB
        - Custom STM32G0 breakout board for ~$3.30 from JLCPCB
    - Step 5: Assemble the PCB

[68] https://github.com/nathancharlesjones/Embedded-for-Everyone/wiki

- “Resources”

The repository includes sub-module links to the five “getting started” guides (with their 14 total example projects), four custom prototyping PCBs, and two custom MCU breakout boards. The first few levels of the project directory are shown below.

```
./
├── Getting-Started-Guides
│   ├── LPC845-BRK
│   │   ├── Command-line
│   │   │   ├── Blinky_v2
│   │   │   ├── Blinky_v3
│   │   │   ├── OpenOCD_and_CMSIS-DAP.sh
│   │   │   └── OpenOCD_and_J-Link.sh
│   │   ├── NXP-MCUXpresso
│   │   ├── Segger-Embedded-Studio
│   │   └── Getting-Started-with-the-LPC845-BRK.pdf
│   ├── MSP430
│   │   ├── CCStudio
│   │   ├── Getting-Started-with-the-MSP430.pdf
│   │   └── MSP430G2553.fzz
│   ├── PIC16-PIC18-dsPIC33
│   │   ├── dsPIC33
│   │   │   ├── Command-line
│   │   │   └── MPLAB-X
│   │   ├── PIC16
│   │   │   ├── Command-line
│   │   │   └── MPLAB-X
│   │   ├── dsPIC33EV32GM002.fzz
│   │   ├── Getting-Started-with-the-PIC16-PIC18-dsPIC33.pdf
│   │   └── PIC16F18446.fzz
│   ├── STM32G031J6
│   │   ├── Command-line
│   │   │   ├── libopencm3
│   │   │   ├── STM32Cube
│   │   │   ├── Start-ST-Link-GDB-Server.sh
│   │   │   └── Start-ST-Link-GDB-Server_v2.sh
│   │   ├── Keil-MDK-Arm
│   │   ├── STM32CubeIDE
│   │   └── Getting-Started-with-the-STM32G031J6.pdf
│   ├── XMC2GO_XMC1100
│   │   ├── Command-line
│   │   │   ├── Blinky
│   │   │   ├── Blinky_v2
│     │     │     └──── OpenOCD_and_J-Link.sh
│     │     ├──── DAVE
│     │     ├──── Segger-Embedded-Studio
│     │     ├──── VS-Code
│     │     └──── Getting-Started-with-the-XMC2GO-XMC1100.pdf
│     ├──── Compatible-Toolchains.csv
│     └──── Compatible-Toolchains.ods
├──── Printed-Circuit-Boards
│     ├──── Breakout-boards
│     │     ├──── STM32F030F4P6-Breakout-Boards
│     │     └──── STM32G0-Breakout-Boards
│     ├──── Eagle-Design-Rule-Files
│     └──── Prototyping-PCBs
│          ├──── Breadboard-Panelized
│          ├──── Breadboard-Single
│          ├──── Stripboard-Panelized
│          ├──── Stripboard-Single
│          └──── README.md
├──── Supporting-documents
├──── LICENSE
└──── README.md
```

Recommendations for Hobbyists, Students, and Educators

For better or worse, the best way to teach oneself about embedded systems is to find a "getting started" or "beginners" book. The book should match the desired MCU and toolchain as closely as possible, since any change to the toolchain (such as trying to use a slightly different MCU or IDE or SDK than is discussed in the book) is distinctly non-trivial and, except in the rarest of cases, completely outside the realm of "getting started"-like activities. Books that discuss the manufacturer-supplied IDE are likely to be slightly less difficult to get started with initially, since they are almost guaranteed to work with the MCU's SDK, whereas in other IDEs there may be more setup involved. In this vein, the PIC, MSP430, and STM32s have the greatest number of books and community support of the MCUs listed above and will likely be some of the easiest for hobbyists and students to develop with. The MSP430 allows for the most gradual progression (having an IDE and SDK that is almost identical to Arduino as well as excellent integration between its various other IDEs, allowing developers to make a very slow and deliberate transition away from the Arduino toward more advanced development), with a few select STM32 MCUs, the XMC1100, and the SAMD21 coming in close behind (owing to their support inside the Arduino IDE and the possibility of being programmed from such IDEs as Eclipse or VS Code). There is little utility in ranking the MCUs by their specifications, since every MCU has its own unique strengths and I would say, much like Jay Carlson did on his fantastic webpage, “The Amazing $1 Microcontroller”[69], that there is no “best” MCU, only a number of excellent MCUs, some of which are better suited for specific applications than others.

After basic proficiency is reached using the selected MCU and toolchain, the hobbyist or student should next learn how to program their MCU from the command-line and what the vendor-supplied start-up code is doing. Although this process can be rather grueling, acquiring these skills is the best way for the hobbyist or student to understand what all of those myriad IDE options are and it significantly increases the number of tools and toolchains that are accessible to the developer. Even if the hobbyist or student decides to never again program their MCU from the command line or to look at another piece of startup code, having done so once is the best way to understand how every other tool works, which is what allows the developer to actually use these more advanced tools to their fullest capacity.

After that, it is highly encouraged for every developer to experiment with different MCUs and development environments. The developer should change only one thing at a time (MCU, development environment, or SDK) and then try to reach a similar level of proficiency with the new toolchain as was had with the previous toolchain. Not only does this expose the developer to a multitude of MCUs, development environments, and SDKs, but it strengthens the developer’s understanding of how each toolchain works, demystifying toolchain errors and problems.

Lastly, a smattering of random recommendations:

- Take notes as you work, even if all you’re doing is chronicling the things you’re doing. Doing so helps organize your thoughts and becomes an almost invaluable record of the things you’ve tried and failed, tried and succeeded, and learned along the way.
- Always buy components from authorized distributors. The only exceptions to this rule should be components whose specifications you have some means of inspecting and verifying (e.g. a plastic enclosure, a resistor, an LED, etc) or components for which it does not matter if they do not meet their specifications.
- Find a way to give back. Stop by a local makerspace or high school to see if you can offer anyone any pointers. No matter your experience level, there’s always someone with less than you that you can help.

Future Work

In the introduction to this paper, 10 challenges to embedded systems development for hobbyists, students, and teachers were presented. By creating custom development boards and identifying inexpensive MCUs, we’ve attempted to reduce the impact of the “cost and availability of tools and materials”, and by documenting the process of getting started with six of those MCUs and creating a few novel makefiles and scripts, we’ve also attempted to reduce the impact of the “Software download, installations, and configuration/setup” and the “Usability of the tools”. It is the opinion of the author that future work should focus on tackling each of these challenges, perhaps by solving some of the problems below.

Toolchain setup should be made easier by:

[69] Carlson, “The Amazing $1 Microcontroller.”

- requiring few or no dependencies,
- providing the user feedback (e.g. on whether project settings have been set correctly for the target MCU, if the IDE can even recognize that the target MCU is attached, etc), and
- reducing the number of configurable options initially available to the user (e.g. by allowing the user to enter an "easy" or "simple" mode for the GUI or by including an "I don't know" button on all prompts that uses default or "best guess" values when the developer isn't sure how to answer the question).

Texas Instruments is already implementing some of these suggestions (using an online IDE with only one dependency, allowing users in CC Studio to enter a "Simple" mode) and it shows: CCS Cloud was, by far, the easiest IDE to use "out of the box" and the MSP430, in general, is the best MCU that the author can recommend to someone who needs an easy way to transition away from educational platforms. Running an IDE or pre-configured development environment from inside a containerized environment is another way to reduce the number of dependencies; effectively there are none, since the user is never aware of which ones were required for the setup of the development environment.

Developing the embedded systems should be made easier by:

- increasing the number of "minimal resource" development boards that are available, and
- engineering the SDKs to have a sliding scale of difficulty.

Creating additional inexpensive development boards would clearly provide hobbyists with more MCUs to choose from but it would also have the indirect effect of making this development easier by allowing hobbyists to find the MCU that matches a book or tutorial they want to follow along with, instead of the other way around (trying to find a tutorial or book to match the MCU they have in their hands). The problems with trying to find a tutorial to match the MCU can clearly be seen when one is looking for example code to get started with a new MCU; most manufacturers only provide example code to run specifically on one of their (more expensive) development boards and porting the example to the specific MCU one might have with them is not always easy (it definitely is not a "getting started" kind of activity). Additionally, SDKs should be crafted in a way that developers can specify as much or as little about the peripheral they're trying to use as they want, with the SDK filling in the rest as needed. Most SDKs only expose one type of interface: either the "Arduino-style" interface which prioritizes simplicity (e.g. "Serial.begin(9600)", "pinMode(6,OUTPUT)", etc) or one that prioritizes being able to customize each peripheral (often requiring that multiple parameters be set before the peripheral can function). The code generator for STM32 has taken one step forward in this direction, allowing users to specify, by module, whether they would like the generator to use the "HAL" (hardware abstraction layer) library or "LL" (low-level) library. Ideally, however, the API provides a "sliding scale" of difficulty, perhaps by using constructors with default values or providing a series of nested APIs that present an increasing number of options to the developer. In this manner, the "resolution" of the SDKs' difficulty increases from only one or two levels to many, allowing the hobbyist to use exactly the parts of the API that match their skill level.

Put another way, there need to be additional tools that make it easier for hobbyists to succeed and harder for them to fail. The myriad ways a hobbyist can, and often does, fail can be highly discouraging. Like a game, in which the difficulty is only really increased as the player develops additional in-game skills, these tools need to find a way to only be as challenging to the developer as their ability level dictates and to only increase in their complexity as the developer becomes more skilled.

Conclusion

Although embedded systems development is significantly easier and cheaper to do than it was only a few decades before, there still exists a veritable minefield of problems that must be navigated before the hobbyist, student, or educator can successfully participate in it. Ten broad categories of challenges were presented to describe this minefield, from “Cost and availability of tools and components” to “Breadth of domain knowledge required”, which, unfortunately, also includes “Usability of the tools”, meaning that the very things intended to facilitate embedded systems development are sometimes the things that inhibit self-learners. The work conducted in the course of this independent study was intended to help overcome these barriers. Over 24 MCUs in either a DIP or SOIC package and costing less than $8 each (as of the time of this writing) were identified as being suitable for hobbyists and students to develop with. Six of these MCUs were procured and programmed by the author using a total of 11 different toolchains and five different debug adapters. That process was documented with step-by-step instructions and 14 of the completed project files were made available for others to use and follow. Although a comprehensive solution could not be provided, this paper and its accompanying Git repositories will hopefully serve to illuminate that path just a little bit better than it was before.

Appendix A: The Problem of Counterfeit Electronic Components

Counterfeit electronic components proliferate in the market and, by most measures, are only on the rise. Around "1% of semiconductor sales are estimated to be those of counterfeited units"[70] and "as many as 15 percent of all spare and replacement parts purchased by the Pentagon are counterfeit."[71]

A counterfeit electronic component is one whose "material, performance, or characteristics are knowingly misrepresented by the vendor, supplier, distributor, or manufacturer."[72] These could be upclassified or mis-classified electronic components desoldered from e-waste, out-of-range or defective parts discarded from a manufacturing line, or "ghost" parts produced after-hours on the exact same production line as genuine components but without correct serial numbers or lot codes. In 2008, a component distributor named SMT took a trip to Shenzhen, China, and documented extensively some of the counterfeiting practices[73] they found there. If the counterfeit parts are not used in any sort of extreme or critical fashion then they may function like normal and the developer may be none the wiser. However, they may also lead to inexplicable device failures or, as I found out in the course of this study, they may be detected as counterfeit devices by the manufacturer's official software, which (understandably) will prevent them from being programmed from that piece of software. Of course, if the part is being used in any sort of safety-critical applications, there is also the possibility of personal injury or fatalities.

The only guaranteed method of ensuring that you are buying genuine parts is to buy those parts directly from the component manufacturer. Since most manufacturers won't entertain low-volume customers, this excludes all hobbyists, students, teachers, and makers. The next best course of action is to buy electronic components only from authorized distributors or distributors who offer some sort of guarantee against counterfeit components. The following two search engines can help you find genuine parts from authorized distributors:

- ECIA Authorized[74]
- Source Engine[75]

Of course, well-known distributors like Digi-Key, Mouser, Arrow, and Future all show up in searches using these two search engines. Additionally, LCSC[76] is a China-based distributor that guarantees the authenticity of their parts.

[70] Guin et al., “Counterfeit Integrated Circuits: A Rising Threat in the Global Semiconductor Supply Chain.”
[71] “The Committee’s Investigation into Counterfeit Electronics Parts in the Department of Defence Supply Chain.”
[72] Livingston, “Avoiding Counterfeit Electronic Components.”
[73] http://asq.org/asd/2009/03/compliance/counterfeit-parts.pdf
[74] https://www.eciaauthorized.com/en
[75] https://www.sourcengine.com/
[76] https://lcsc.com/

Notably missing from these lists of authorized distributors are Chinese distributors AliExpress (owned by AliBaba) and BangGood. Although clearly not everything sold on those websites is counterfeit, in all of my research I found the following to be true: "If a deal appears to be too good to be true, it probably is." Unless you have the means at home of verifying a part's authenticity and unless the possibility of getting a counterfeit part is of no consequence to you, then I strongly recommend that you only buy your components from one of the other distributors listed above.

Two interesting case studies about both the positives and negatives of unknown parts from China are linked below.

- A $17 Power Supply: Is it Worth it?[77]
- ESP32 Development Board - Official vs Clone[78]

[77] https://www.digikey.com/eewiki/pages/viewpage.action?pageId=90243471
[78] https://www.hackster.io/rayburne/esp32-development-board-official-vs-clone-7f4ff7

Works Cited

"Analysis of the Global Microcontroller Market." Frost & Sullivan, August 1, 2013. https://cds-frost-com.prox.lib.ncsu.edu/p/58859#!/ppt/c?id=NBD8-01-00-00-00&hq=%22Analysis%20of%20the%20Global%20Microcontroller%20Market%22.

Analytis, Santhi, Joel A. Sadler, and Mark R. Cutkosky. "Creating Paper Robots Increases Designers' Confidence to Prototype with Microcontrollers and Electronics." *International Journal of Design Creativity and Innovation* 5 (October 28, 2015): 1–12.

"ASPENCORE 2017 Embedded Markets Study." EE Times, April 2017. https://m.eet.com/media/1246048/2017-embedded-market-study.pdf.

Barnes, Mike, M Bailey, P R Green, and D A Foster. "Teaching Embedded Microprocessor Systems by Enquiry-Based Group Learning." *International Journal of Electrical Engineering & Education* 43, no. 1 (January 2006): 1–14.

Barrett, Steven, Jeffrey Anderson, Jerry Hamann, Robert Kubichek, Suresh Muknahallipatna, John Pierre, David Whitman, and Cameron Wright. "Embedded Systems Design: Responding to the Challenge." In *2009 ASEE Annual Conference & Exposition*. Austin, Texas: ASEE, 2009. https://peer.asee.org/embedded-systems-design-responding-to-the-challenge.

Committee on STEM Education. "Charting a Course for Success: America's Strategy for STEM Education." National Science and Technology Council, December 2018. https://www.whitehouse.gov/wp-content/uploads/2018/12/STEM-Education-Strategic-Plan-2018.pdf.

Corno, Fulvio, Luigi De Russis, and Juan Pablo Saenz. "On the Challenges Novice Programmers Experience in Developing IoT Systems: A Survey." *The Journal of Systems and Software* 157 (2019). https://doi.org/10.1016/j.jss.2019.07.101.

Devine, James, Joe Finney, Peli de Halleux, Michal Moskal, Thomas Ball, and Steven Hodges. "MakeCode and CODAL: Intuitive and Efficient Embedded Systems Programming for Education." *Journal of Systems Architecture* 98 (September 2019): 468–83.

El-Abd, Mohammed. "A Review of Embedded Systems Education in the Arduino Age: Lessons Learned and Future Directions." *International Journal of Engineering Pedagogy* 7 (June 2017): 79–93.

Guin, Ujjwal, Ke Huang, Daniel DiMase, John M. Carulli, Mohammad Tehranipoor, and Yiorgos Makris. "Counterfeit Integrated Circuits: A Rising Threat in the Global Semiconductor Supply Chain." *Proceedings of the IEEE* 102, no. 8 (July 15, 2014): 1207–28.

Hlubinka, Michelle, Dale Dougherty, Parker Thomas, Stephanie Chang, Steve Hoefer, Isaac Alexander, and Devon McGuire. "Makerspace Playbook: School Edition." Maker Media, Spring 2013. https://makered.org/wp-content/uploads/2014/09/Makerspace-Playbook-Feb-2013.pdf.

Kurti, R. Steven, Debby L. Kurti, and Laura Fleming. "The Philosophy of Educational Makerspaces, Part 1." *Teacher Librarian*, June 2014. https://pdfs.semanticscholar.org/c1c9/1df674af209b768853efebed8764324b4698.pdf.

Lahtinen, Essi, Kirsti Ala-Mutka, and Hannu-Matti Jarvinen. "A Study of the Difficulties of Novice Programmers." In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, 14–18. Association for Computing Machinery, 2005. https://doi.org/10.1145/1067445.1067453.

Livingston, Henry. "Avoiding Counterfeit Electronic Components." *IEEE Transactions on Components and Packaging Technologies* 30, no. 1 (January 4, 2007): 187–89.

Mondragon, Antonio Francisco, and Adriana Becker-Gomez. “So Many Educational Microcontroller Platforms, So Little Time!” San Antonio, Texas: ASEE, 2012. https://peer.asee.org/so-many-educational-microcontroller-platforms-so-little-time.

“News Release: Microchip Technology Completes Atmel Acquisition and Provides Update on Its Fiscal Fourth Quarter 2016.” Microchip Investor Relations, April 4, 2016. https://www.microchip.com/pdf/mchp-completes-atml-acquisition-and-provides-update-on-its-q4fy16.pdf.

“NXP Acquires Freescale, Becomes Top MCU Supplier in 2016.” IC Insights, April 27, 2017. https://www.icinsights.com/data/articles/documents/970.pdf.

“Press Release: NXP and Freescale Announce Completion of Merger.” NXP Investor Relations, December 7, 2015. https://investors.nxp.com/news-releases/news-release-details/nxp-and-freescale-announce-completion-merger.

Sadler, Joel, Lauren Aquino Shluzas, and Paulo Blikstein. “Abracadabra: Imagining Access to Creative Computing Tools for Everyone.” In *Design Thinking Research*, 365–76. Understanding Innovation. Springer, Chem, 2017. https://doi.org/10.1007/978-3-319-60967-6_19.

Sadler, Joel, Lauren Shluzas, and Paulo Blikstein. “Building Blocks in Creative Computing: Modularity Increases the Probability of Prototyping Novel Ideas.” *International Journal of Design Creativity and Innovation* 5 (February 2016): 1–17.

Severance, Charles. “Massimo Banzi: Building Arduino” 47, no. 1 (January 2014): 11–12.

Staver, John, Naiqian Zhang, Masaaki Mizuno, Gurdip Singh, Mitchell Neilsen, and Donald Lenhert. “Encouraging Interest In Engineering Through Embedded System Design.” Salt Lake City, Utah: ASEE, 2004. https://peer.asee.org/encouraging-interest-in-engineering-through-embedded-system-design.

“Strategic Analysis of the Global Microcontrollers Market.” Frost & Sullivan, January 2010. https://cds-frost-com.prox.lib.ncsu.edu/p/58859#!/nts/c?id=N714-01-00-00-00&hq=%22Strategic%20Analysis%20of%20the%20Global%20Microcontrollers%20Market%22.

“The Committee’s Investigation into Counterfeit Electronics Parts in the Department of Defence Supply Chain.” Committee on Armed Services, United States Senate, November 8, 2011. https://www.govinfo.gov/content/pkg/CHRG-112shrg72702/pdf/CHRG-112shrg72702.pdf.

“The McClean Report Contents and Summaries,” 2016. https://www.icinsights.com/services/mcclean-report/report-contents/.

Weiner, Steven, Micah Lande, and Shawn Jordan. “What Have We ”Learned” from Maker Education Research? A Learning Sciences-Base Review of ASEE Literature on the Maker Movement.” *American Society for Engineering Education*, 2018. https://peer.asee.org/what-have-we-learned-from-maker-education-research-a-learning-sciences-base-review-of-asee-literature-on-the-maker-movement.pdf.

White, Elecia. *Making Embedded Systems*. O’Reilly Media, 2011.