A Skills List for Developing Embedded Software

Dale Word
Dept. of Electrical and Computer Engineering
California State University, Chico

1. What's Unique About Embedded Development
   The very nature of embedded systems dictates that the development of software for them is different than software for general purpose computers.

   a. Background
   The key factors that separate embedded computer systems from other type of computer based systems are as follows:

   - Dedicated Function – Embedded systems are designed to perform one specific set of functions, typically as part of a larger device or system. This is in contrast with a typical desktop system that is a general purpose computing platform, designed to support a variety of applications, each performing a different set of functions.

   - Limited Resources – Due to the fact that embedded systems are typically a component of a larger system, rather than a standalone computing device, they commonly are designed with limited computing resources. This can be the result a desire to reduce production costs, or as a result of the operating environment of the end system. These limitations include the lack of graphical user interfaces, limited storage devices, limited processing power, etc.

   - Hardware Interaction – In general purpose computing systems, the details of the operating environment a typically abstracted by different operating system layers, to enhance portability and interoperability. The limited resources and specific nature of embedded systems prevent the use of those same layers, requiring the embedded software to interface to the operating environment much more directly.

   - Real-Time Processing – The high level of integration with additional devices and equipment in embedded systems frequently dictate that they perform their processing in a fixed time interval. The "real-time" aspect of the application adds additional complexity to all phases of the development process, from design to testing.

   Defining the characteristics of embedded systems is very inexact, because embedded systems vary widely in their configurations and capabilities. There is a full spectrum of systems that are considered embedded systems, from the smallest 8 bit microcontroller system, to the highest end, high performance multiprocessor based multitasking system.

b. Differentiating Factors – Development
The unique set of characteristics that define embedded systems result in an equally unique set of development considerations.

    i. Resource Limitations – Developing code in a resource limited environment requires developers to consider not only the functional aspects of their code, but also its resource usage. This includes memory usage, processor loading, data representation, etc.

    ii. Hardware Interaction – The low level, hardware intensive code that is required in many embedded systems requires an intimate knowledge of the hardware components being used, and the techniques required to access them directly.

    iii. Real-Time – The addition of temporal requirements to any set of software can multiply the complexity of the implementation. The need to consider execution time and behavior patterns significantly changes every aspect of the implementation process.

    iv. Development Cycle - In software development efforts for general purpose computers, the target hardware platform is well-known, fixed, and well defined. In embedded systems this is frequently not the case. The target hardware can vary from a sophisticated, well defined, off-the-shelf platform to a completely untested prototype design. In the cases where hardware development is parallel to the software development, the potential for hardware problems requires that embedded developers consider a much more complex set of possibilities when problem solving. If the software development cycle precedes the hardware development, the software set may need to be developed on a prototype platform, and ported to the actual target when it becomes available.

2. Historical Perspective on Software Development
Software development, in years past, required many of the skills that embedded development continues to require today. Historically, operating systems and development environments were not sophisticated enough to eliminate the need for the low level skills of an embedded developer. As the technologies have improved, there has been a movement toward more and more abstraction of the underlying hardware and operating system. This increasing abstraction of the target hardware, combined with increasingly sophisticated development tools, has given rise to a growing number of software developers that do not possess the skills required by embedded development. Graphically based, integrated development toolsets have only

exacerbated the problem, allowing “point and click” development cycles, that require little or no understanding of what is going on below the application level. In fairness, it should be stated that this trend toward abstraction and tool sophistication has yielded great gains in productivity for general software development. Unfortunately, these gains have come at the cost of losing the skill base that applies directly to embedded development.

3. Detailed Skills List
   The development of embedded software applications requires a skill set that focuses on these unique characteristics of embedded systems.

   a. Architecture Concepts
      One of the key areas of knowledge that is required for embedded development is an understanding and awareness of the underlying computer architecture of the target system.

      i. Memory Map View
         Embedded developers need to have a very “physical” view of their applications. This includes an understanding of the overall memory layout of the target system, and where each component of their application resides in that memory. This view can be obtained by a quick study of the memory map (generated during the link process). It should include an understanding of where code and data segments exist, where stack, heap and constant storage is allocated. This “physical” view, like many of the other skills described in this section, is more critical in systems that have limited memory resources, and less critical in systems with more memory and more sophisticated operating systems.

      ii. Stack Operations
          One of the best ways to gather information about the current state of execution in a multitasking embedded system is to take a snapshot of the stack, and use it to determine the sequence of calls that have led to the current state. In order to do this, a developer must have a good understanding of the way exception processing and context switching affect the contents of the stack, and the structure of the stack frames and context sets that are pushed onto the stack.

      iii. Instruction Set Architecture
           An understanding of the target processor details is another important part of an embedded developer’s skill set. This should include things like: Register Sets, Processor Modes, Addressing Modes, Processor Status Registers, etc. This set of skills may not be required in most development cycles, but can be crucial in resolving subtle problems during debugging, or when implementing

advanced features that directly rely on some unique aspect of the processor.

iv. Exception Handling
A basic understanding of exception and interrupt handling is important in dealing with external events or data sources, and error conditions that result in a processor exception. This level of processing is typically handled for, and hidden from the developer in more sophisticated operating system and toolset systems.

b. Programming Skills:
The range of programming skills and the different areas of focus can be viewed as a spectrum, from the most abstract, high level programming environments to the lowest level, most hardware specific environment. While development for general purpose machines typically focuses on more abstract, high level programming concepts, embedded developers must also focus on low level, more hardware specific details. Figure 1 depicts this relationship graphically.

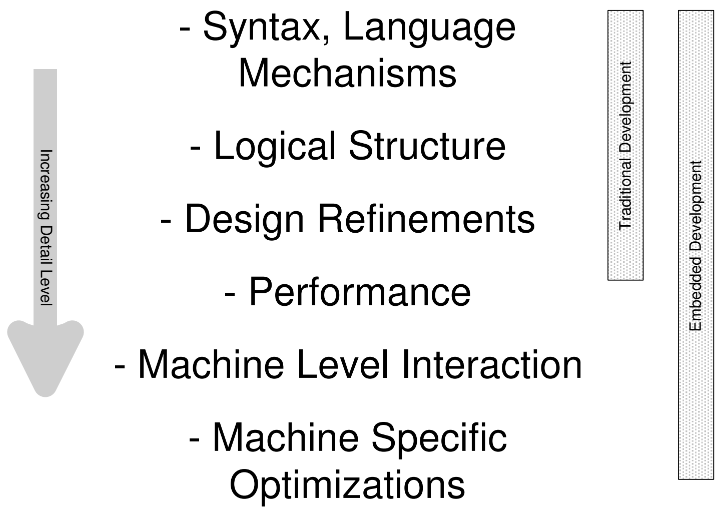

Figure 1

i. Assembly Language Programming
Assembly language code historically has been an important part of many embedded systems, providing better performance and a smaller footprint for critical sections of code. With the advances in processor speed and memory size relative to cost, this is becoming less and less necessary. A basic knowledge of the assembly language of your target processor is important, but more of a background topic than a central development skill as it was in the past.

ii. Low Level C Programming

One of the key characteristics of embedded software is its low level, direct interaction with hardware. To implement these functions in C, there are a few specific aspects of the language that need to be understood:

- Direct memory access using type casting of constant values

```
*(int *)0xFFF0000 = some_value;
status = *(int *)0xFFF00000;
```

- Bit level operations, including masking and shifting

```
*(int *)UART_STAT_REG = uart_status & TX_EN_BIT &
RX_EN_BIT;
```

While these are not new concepts to experienced C programmers, they are commonly misunderstood or overlooked by developers coming from a higher level, more abstract programming background.

c. Tools/Environment

The environment used to develop embedded software is sometimes the thing that is most different from other types of software development.

i. Headless Targets

The very nature of an embedded device that doesn't have any kind of external display or interfaces can make getting debugging information difficult. If you have sophisticated tools, this is usually not a problem. For the rare case where you don't have support form tools, some cleverness and creativity may be required to gain visibility into the embedded application. This may involve using hardware that was not intended to communicate data to provide rudimentary information access.

ii. Cross Development

One of the key aspects that helps define embedded development is the pattern of cross development – creating code on a machine of one type, to run on a machine of a different type. Once this "foreign" executable has been created, it must be loaded onto the target hardware. This involves several concepts that are important for a developer to understand. First, a developer needs to understand what the boot sequence of the target device really is. Typically the device will boot to a monitor program in ROM, initialize some communications interface, and then wait to receive an executable code image from the development machine. Understanding this sequence, and where each of these components resides in the overall memory map is essential for a developer to have a clear picture of the target board's behavior.

iii. Hardware Knowledge
The close interaction with hardware in embedded systems requires that developers understand the characteristics and capabilities of each external (to the CPU) hardware component being accessed. This usually means studying databooks and reference manuals for the specific chips involved, and developing an understanding of the configuration and operation of each of them. This is an area in which example code sets can be of great value in helping speed up the learning process. Looking at an existing driver for the chip you're using can often yield great insights into its use.

d. Problem Solving
One of the things that separates embedded development most from general purpose systems development is the unique set of problem solving skills it requires.

i. Real-Time Debugging
The addition of real time constraints to an application not only complicates the design and development process, but can make the debugging process extremely complex. The key difference is the need to maintain an intuitive model of the temporal aspects of the application. This includes recognizing the different behaviors that result from timing dependencies, resource conflicts, interrupt processing, etc.

ii. Understand Your Toolset
Starting out with any new embedded development tool set can be a trying process. Most modern tools sets provide a wealth of information and features, but they require an investment of time to learn the details of how to make use of these capabilities. The time spent learning a toolset almost always pays off later in time saved during debugging. Besides learning what a toolset CAN do, it is also crucial to understand its limitations. Examples of this include understanding the effects of remote debug tasks running on your target systems, or pipeline behavior and its affect on emulator performance.

iii. Holistic Problem Solving
The term *holsitic* is defined as " views in which the individual elements of a system are determined by their relations to all other elements of that system". A holistic approach to problem solving involves looking at all the system components, and objectively analyzing the data to determine the root causes of a problem. The holistic nature of this approach means that the interactions of all components (hardware, RTOS, application software, etc.) are considered together to identify the causes of unwanted behaviors.

iv. Vertical vs Horizontal Problem Space
When considering the components that make up any computer system, one way to view the overall picture is to look at it as a hierarchy, layered from the most concrete (hardware) to the most abstract (user interface). Given this view as a backdrop, it is useful to look at the number of layers that are incorporated into a typical training or problem solving exercise.
For most formal academic programs, the focus is on doing very detailed work on one or two layers, taking a "horizontal slice" of the overall problem space. This approach, while giving a great deal of exposure to one specific layer of the problem space, fails to model the way difficult problems are solved in real commercial environments.

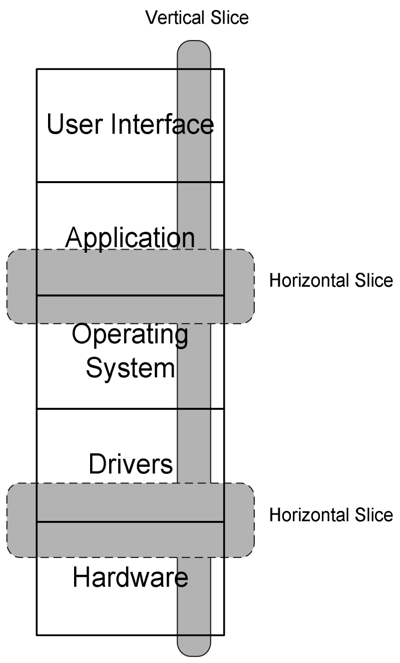

Figure 2

v. Given a difficult or subtle problem, it is often required to take a top to bottom view of the problem, or a "vertical slice" of the problem space. This ability to seek solutions across functional boundaries, and "drill down" through the layers of a system is characteristic of embedded systems development.

vi. Problem Solving Methodology
Arthur Clarke, the famous science fiction author and futurist, once said: "Any sufficiently advanced technology is indistinguishable from magic." I would propose that there is an embedded corollary that says: "advanced technology produces problems that ARE indistinguishable from magic (at first)". The following is a list of problem solving steps:

1. Find Clues – Like NTSB sifting through the wreckage of a plane crash – examine each small piece may have some significance. Look at stack traces, buffer contents, etc., identifying anything that indicates unexpected behavior.
2. Gather as much information as possible – The solution often lies not in some complex implementation detail, but rather in taking a simple (and somehow different) view of what is occurring.
3. Pursue Theories –Like walking a maze, pursue each path until you're sure it has been exhausted. When that path has been exhausted, backtrack and start pursuing another one.
4. Find the answer – Don't assume that making some fairly arbitrary change that makes the problem go away is an

adequate solution. Find the root cause, so you can be sure it is completely eradicated from the system.

5. Logic is your main tool – No matter how much pressure you're under, hang on to a logical, methodical approach. Be careful to distinguish between what you know (and have proven), versus what you're assuming.

4. Transitioning to Embedded Development
   For any organization transitioning to embedded development for the first time, the first, most critical resource is the development staff. Creating an embedded development team means either hiring the necessary talent or providing specific training for current developers who may lack the specific skills required for embedded development.

   a. Hiring - Degree programs
      Obviously, if the situation permits, hiring experienced embedded developers is preferable. While evaluating the industry experience of a candidate and its applicability to any development effort is too specific to discuss in general terms, the applicability of different engineering degree types and their applicability to embedded development can be briefly summarized.

      i. Computer Science
         In the past 20 years or so, the focus of Computer Science degree programs has changed. In the early days of the computer industry, Computer Science programs tended to teach the fundamental concepts associated with software theory, development and design – without addressing specific application areas as major topics. As the industry and technologies have matured, these application areas have become recognized sub-disciplines of Computer Science. This has led to more concentration on these more abstract areas of study, with a reduction in the amount of time spent addressing low level, concrete details. For embedded development, this means that some Computer Science graduates may lack adequate exposure to things like assembly language programming, interrupt and driver level code, etc. The positive aspect of this trend toward abstraction is that many of these same graduates may be more highly skilled in design approaches, leading to higher quality code once they overcome any low level shortcomings they may have.

      ii. Electrical Engineering
          On the opposite end of the spectrum is the hardware-focused discipline of Electrical Engineering. Just as in Computer Science, the evolution of the industry has led to more diversity and specialization in this discipline also. This trend requires an

increased focus on these Electrical Engineering specific topics, and less on software related concepts. One development that may help prepare Electrical Engineering students for a transition to more software oriented tasks is the growth in the use of high level logic design languages. Experience with these tools, even in the context of circuit design, provides some fundamental knowledge that can ease the transition into code development later.

iii. Computer Engineering
Given the constant growth in the amount of information that must be included in Computer Science and Electrical Engineering programs, it follows that these programs would need to spend more time addressing those topics firmly within their disciplines, leaving less time to address topics in the other discipline. As these programs migrate toward their ends of the spectrum, the growing void is filled by the somewhat younger discipline of Computer Engineering. Focusing on both hardware and software concepts, it clearly addresses the issues associated with embedded development

iv. Finding the Right Fit
Figure 3 shows a graphical view of this degree program spectrum.

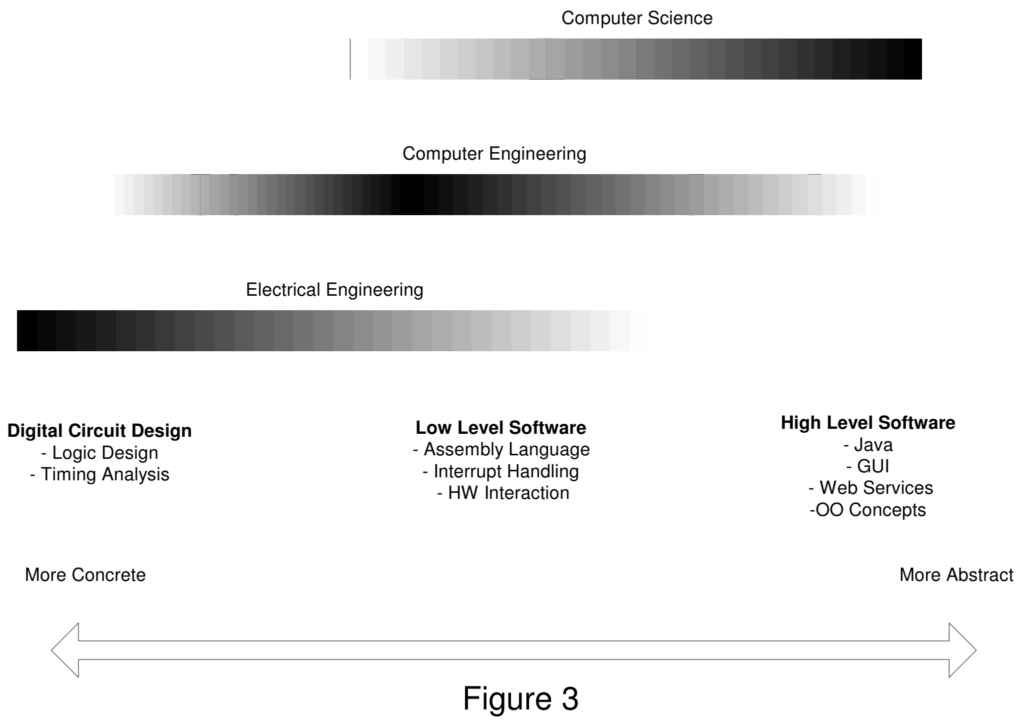

Figure 3

The key to finding the best candidates for a given organization is to identify where in this spectrum of knowledge skills are needed, and finding candidates whose backgrounds match up.

b. Professional Training

For existing employees without a background in embedded development, or for those needing a refresher course, some professional training may be required.

i. Extension/Remote Classroom Programs
With the growth in remote classes being offered via the internet, many options exist for acquiring embedded development education in this manner. The following is a brief list of some of these programs:

Embedded development certificate programs:
University of California, San Diego
University of Colorado at Boulder
University of Washington

Embedded development courses offered:
University of California, Berkeley
University of California, Irvine
Iowa State University

(See the references at the end of this document for web links to each of these programs)

ii. Mentoring
A more informal means of providing training is to establish a mentoring program, using more experienced developers to train their less experienced peers. One way to accomplish this is to have the trainee "shadow" the mentor through a full development cycle. If there are no experienced mentors available in-house, it may be beneficial to hire outside consultants to lead an early development cycle, while acting as mentors to the permanent development staff.

iii. Self Teaching
A final option, if no other resources are available, is to explore self-teaching exercises. The most common way to do this is with the aid of a reference text. This text should address general embedded development concepts, and provide a set of examples to work through, on some form of target hardware. This kind of exercise can provide a good initial hands-on learning experience. The problems with this approach are 1) help can be difficult to get when you get stuck, and 2) they're usually based on fairly simplistic, low cost target hardware, restricting the complexity of the exercises. (See the book references at the end of this document for some title suggestions)

5. Tips/Hints
   The following is a list of informal tips for getting started with, and getting through your first embedded development effort:

   a. Start with reliable target hardware – The last thing you need during your first embedded project is to have more problems and variables introduced by the hardware. If your final target is a custom board, start with a similar evaluation board until you get past the early stages of the software development process.

   b. Do not overestimate the value of demo code – Getting someone else's code to run, no matter how great of a demo it is, is not the same as generating your own working code from scratch. There may be several issues to be resolved in the process of getting your code to actually run on the board.

   c. Start simple – Get it to do something simple first. Take small steps early, to verify your basic assumptions, before you start adding complexity.

   d. DO NOT expect vendors to design your product – Most vendors of embedded development products provide great support, often with very experienced embedded developers. Be careful not to ask them to design your product for you, 1) it's not their job – they support their product, not everything that can be produced with it, and 2) you'll probably get a design that looks more like their latest application note, rather than a good solution for your product.

   e. DO expect vendors to solve their own problems – When you encounter problems with a tool, and the answer is not obvious in the documentation, use the vendor's support services. You may be struggling with a problem they've seen before. Be prepared to isolate the problem section of your code by generating a "cut-down" version of the code – this will greatly simplify the process of identifying the true cause of the problem.

   f. Keep a test/development log – Embedded systems, particularly real-time embedded systems, can exhibit complex and subtle behavior patterns. Over the span of a lengthy development cycle, it is difficult to track the cause-effect relationship between changes to the code and the resulting effect on behavior. A simple way to maintain a persistent record of this is to keep a log, with entries for every set of code changes, the date, and the test results generated. This does not have to be anything rigid or formal, but rather a set of notes, tailored to each developer's personal style. Using some form of version control software to manage the code base is also very helpful, yielding a precise record of every change to the software, in chronological order.

g. No matter how mysterious a problem seems, be methodical – Resist the temptation to make arbitrary changes to see if you can make an illusive problem go away. This will probably just cover up a problem that will come back to haunt you later. [Also: No matter how mysterious or bizarre the problem seems, don't start sacrificing chickens or making offerings to the digital gods – it won't help, and it looks vaguely unprofessional.]

6. Conclusion - Defining Your Needs
There is no single definition of the skill set required to begin embedded software development. The answer varies with each different development configuration. Factors like the complexity of the target system, the time and budget constraints involved, and the background of the existing development staff all contribute to the unique character of each development effort. The skills needed to develop embedded code for a simple, 8-bit microcontroller based product are distinctly different than those required for a high-end real-time system with multiple 32-bit processors. The key is to identify what your specific needs are for your development, and ensure that you satisfy those needs with the appropriate staff and equipment.

7. References

a. Books
"Programming Embedded Systems in C and C++", Michael Barr, O'Reilly

"Fundamentals of Embedded Software: Where C and Assembly Meet", Daniel Lewis, Prentice Hall

"MicroC/OS-II, the Real Time Kernel", Jean Labrosse, CMP

b. Websites

Embedded development certificate programs:

University of California, San Diego
http://www.extension.ucsd.edu/Programs/certificate_directory_su00.html

University of Colorado at Boulder
http://ece-www.colorado.edu/embedded.html

University of Washington
http://www.outreach.washington.edu/extinfo/certprog/emb/emb_main.asp

Embedded development courses:

University of California, Berkeley
http://www.unex.berkeley.edu/eng/#8

University of California, Irvine
http://www.unex.uci.edu/cgi-bin/order/scan/sf=catno/se=214

Iowa State University
http://www.lifelearner.iastate.edu/courses/disciplinesp03.htm