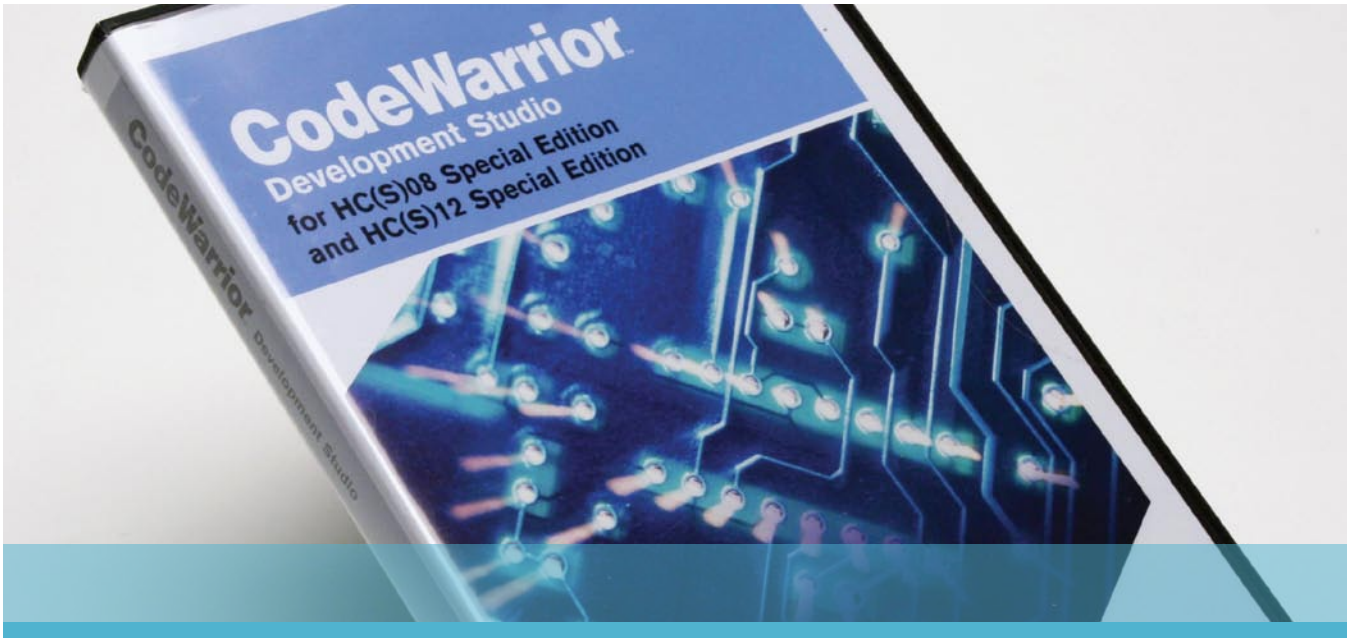
Robert Lewis—Design Alliance Partner

The Path to Success
Embedded processors and C



The project began as projects often do—with a reasonable marketing plan, a functional specification and a realistic time line. And it probably would have all worked exactly as planned if the inevitable had not happened. The predictable change of scope… "we need to add would it be possible to, we could really improve the product if we just added, it can't be that hard. After all, it's just software."

But, let's go back to the beginning—before the amoeba evolved into the eagle. The project was simple enough, take in a frequency, convert it to another based on a user-selected setting and output it under certain conditions. It all could have been done with a simple 8-bit micro, but we wanted extra debugging capabilities and at that time the 908 had only one break point, so we chose the HCS12 family. We didn't want to program in assembler, so C seemed the only reasonable alternative and a good solution. Assembler would have worked fine, but the overhead in managing consistent programming standards and sorting out debugging because of naming problems and register addressing errors was not worth the extra risk.

Next, we started a search for a good ANSI compiler. We looked at the CodeWarrior® tool set, but given the size of the project and the limited functions we would be performing, it didn't seem reasonable to spend a significant amount of the budget for the tool set and the large amount of documentation with seemingly endless switches and options. This suggested a lot of time would be spent just learning the interactive development environment (IDE).

So we began to look into free and third-party C compilers. The free compilers seemed to be stable enough, but it became readily apparent that a lot of expertise would be required to set up the system for compiling and linking, and the only support was through Internet forums. At that point in time, we didn't have either the time or background to take the risk with that many unknowns, so we focused on what seemed to be a good third-party alternative product. The users in the forums said great things about the stability and support of the full ANSI C compiler, so we settled on using the chosen product.

Now that we had the compiler, the debugger was next. It would have been nice to buy a $20,000 in-circuit emulator (ICE), but that didn't seem like a reasonable fit with our $300 compiler or the budget, so we ruled it out rather quickly. Instead, we looked into a product that required "no in-circuit emulator." After running the product in evaluation mode for a few days, I knew we had made a good decision. The product had an excellent and well thought out interface, was solid and had no crashes,

hang-ups, dumps, lockups or other usual issues. (The latter features we are all aware of in the wonderful world of “bling” web pages and little substance.) We could see this product was going to work well.

We had just a few more tools to put together before we could start. We needed a multi-pane editor, a source code repository to track revisions and a source code analysis system for cross referencing and finding symbols. The open-source product for source code analysis was an amazing free product with a lot of functionality but, alas, no documentation.

By now you may see the evolution toward what we failed to see. We were effectively building our own IDE and therefore would also be responsible for integrating and supporting every aspect. Where there was no documentation, we would be required to discover how to make the tool work. This proved to be no small task.

As expected, it took a while to get the compiler and debugger to work. The debugger was fine, but this was a new release for the compiler and there was a lot of hair pulling trying to get the object code to flash correctly and matching up correct access to the paging register. Unfortunately, this problem occurred while the compiler vendor was away on a much needed vacation. In the end, we had good support from the debugger group. They sorted out the inconsistencies in the compiler and when the vendor returned, the compiler was changed to flash and loaded correctly.

For a time, all went well, the code was in design and the coding standard was more or less established with respect to naming conventions, function calls and so forth. Then, the first change came along. Instead of a simple BCD switch, we now needed a four line by 16 character LCD display and a push button to select the options. (“What options? A BCD switch doesn’t have options… it has switches! They are on or off and they are read at startup! Well we need a display and a switch…”)

Now we had to write routines to set up the LCD control registers, strobe the address and data, write all the primitives to position characters, build lines from characters and about 40 other display related routines. Having done that, an interrupt handler had to be added for running the five-way switch. Additional routines were needed to track where the cursor was on any given line within the menu. There were now sets of lines within sets of functions, so we needed a menu to keep track of the options.

The code had grown, but was still manageable until the next request came along, which was a way for users to save their options. My response was, “if you use a BCD switch, you don’t need to save anything, you just look at the switch.” Since that suggestion was rejected, we moved on at first to a serial data link. But now we also required a PC application. After bread boarding a serial link and working with a sophisticated fourth-generation tool for the PC application, I knew the inherent error-prone connectivity of serial transfer was not for me. The system would hang if it got out of sync, or if it miscued a byte everything would get out of step and do bizarre things. The only solution would be a significant amount of handshaking software. After much casting about, we decided to add an SD card with a FAT file system rather than use an active link to the PC. This is where everything took a turn for the worse.

The SD card addition added about 20 KB to the object output and a significant problem appeared. The debugger started showing the code stepping off into regions that were completely unrelated. At seemingly random times, the execution would jump into unrelated functions. We spent days writing code to try to trace possible stack problems and more time trying to trap what could possibly be errant interrupts, all without success. Finally, with the help of the author of the debugging tool, we determined that the compiler was producing incorrect symbol tables and linkages because of the order of the include functions and because the functions were included in-line rather then linked in. The vendor of the compiler gave us a work-around and we were back in business again, albeit somewhat worse for the wear. We trudged on waiting for a permanent fix, but at least we were able to proceed.

Next came the statement that the character display was “under whelming” and devalued the product—we needed a graphics display. Nothing too fancy. It could be simply monochrome, something that was not as crude and archaic as our current character display. My reply that a “BCD switch was stylish, compact and came in a variety of colors” was taken as neither constructive nor helpful.

The addition of the graphics routines with the font tables added another 20 KB to the object code and further problems appeared in the code execution. The compiler appeared to be having problems with setting the paging register. The vendor of the compiler was dedicated to doing his best to support the product, but it was also apparent from cross posts in the forums that his company was devoting their time to a new compiler for a different manufacturer’s product.

This was the last straw; we had to finally admit that if we were going to maintain the code base we had developed, we needed a product that had guaranteed stability. The only option was to rethink using the CodeWarrior tool set. We were not naïve enough to think that the CodeWarrior IDE would be without its own particular brand of problems, but we did know that large multinational companies like Freescale that made the processor

and provided the tool set would have a very strong motivation to keep the tool set current and operational.

So, we bought the full version of the CodeWarrior IDE but without the full version of Processor Expert™ beans. This version allowed us to build any size of code base and it also had some basic Processor Expert objects. At the time, we knew nothing about the Processor Expert beans. It seemed that just mastering the CodeWarrior tool set would be enough to start with.

I went through some of the training modules on the Freescale Web site, and the process seemed less daunting than I had first imagined. Next, I started to convert the code. I never adjusted any of the compiler or linker switches but used the defaults.

In about two days, we had the code compiling and linking without error. There were very few changes since the compiler we had used was ANSI-compliant. The main changes were in the file system naming, pragma statements and building against the CodeWarrior definition files for the port and ECT assignments. The file system had used a few reserved function names such as fopen( ), fclose( ) that were found in the ANSI library. They were easily corrected by renaming our function calls to fopen_imn, and so on. The pragma changes were obvious and very simple to correct.

We loaded the code and ran it, and to our amazement it worked perfectly—the first time. Not only that, but because the CodeWarrior IDE was an optimizing compiler and our old compiler was not, the CodeWarrior code was more than 30 percent smaller and significantly faster. All of the problems and bugs we encountered in our previous code base disappeared. And, as a bonus, our old friend the (no in-circuit emulator) debugger worked with the CodeWarrior ELF output. We could use either the CodeWarrior debugger or our previously chosen tool to debug. We currently use the X-Gate processor; therefore we most often need to use the CodeWarrior debugger.

In hindsight, what can be learned from all of this? Even though I had been doing very large control system projects for more than 30 years, I was still caught by the trivial traps I had often cautioned others against. When it was not my money, I bought the best tools and whatever else was needed to ensure the fastest project completion with the least risk. When it was my money (partly), I traded my time for a lower cost tool set because I trusted my expertise to make it all work; and it did… up to a point. But there comes a point at which you cannot control the outcome of a project when others are involved. If I were to calculate my time at even a very low rate, we paid for the CodeWarrior tool set several times over. Additionally, that doesn't account for the emotional expense or the lost revenue due to the project being late. Sometimes, it may not be an option to come up with the money for tool set. However, now with the free CodeWarrior tool set supporting up to 32 KB program sizes and with a compiler that optimizes so well, a lot of projects can be built that would previously not have fit. Later, once the project is large enough, there may be the funds to afford the extra expense.

When you are dependent on a small vendor with a small customer base, that vendor can be influenced by factors that they cannot control. The loss of a key employee, sickness and change in revenue base, for example, can remove any vendor's ability to deliver.

This may not be a factor worth considering in all projects. Whether this plays a part in your decision process depends on the end user. If your product is one-of-a-kind and a personal customer that you can support locally and quickly with a work around, there may be no risk at all. You may never encounter compiler and linker bugs, either because your code base is small or so thoroughly tested from past projects that the new code is easily debugged and separate. However, if you are building a product that will go into a real-time control system or be manufactured in any sizable volume, finding an error once the product is with the end user could cause a sizeable revenue consequence and loss of credibility. This type of project needs an immediate response, and it may be wise to rethink your "insurance policy." That is, when you pay the extra money for a proven, supported product, you can escalate the support problem and get a response. This may not solve the problem immediately, but it's better than being told "gee we've never seen this before; check on the forum to see if anyone else has a suggestion." That is a very lonely feeling in a crisis.

Did I mention our journey through the twilight zone of Processor Expert beans? No? Well, perhaps another time.

Robert Lewis is an engineer at iMn MicroControl Ltd. He holds Bachelor of Science and Master of Science degrees in electrical engineering, with a specialty in microprocessor-based systems.