# Porting a Network Cryptographic Service to the RMC2000: A Case Study in Embedded Software Development

Paolo de Dios and Stephen Jan
Dept. of Computer Science
Columbia University, New York, NY
{pd119, sj178}@columbia.edu

***Abstract***

*This paper describes our experiences with porting a TLS cryptography service to an embedded microcontroller. We describe some key development issues and techniques involved in porting networked software to a connected, limited resource device such as the Rabbit RMC2000. We examine the effectiveness of a few purported porting strategies by examining important program and run-time characteristics.*

## 1. Introduction

Embedded systems present a different software engineering problem. These types of systems are unique in that the iron and the code are tightly integrated. The limited nature of an embedded systems operating environment require a different approach to developing and porting software. In this paper, we discuss the key issues in developing and porting a UNIX system level TLS service to an embedded microcontroller. We discuss our design decisions and experience porting this service using Dynamic C, an ANSI C variant, on the RMC2000 microcontroller from Rabbit Semiconductor. We examine some performance comparisons to ascertain the feasibility and effectiveness of some common programming idioms for embedded software development.

Porting software across platforms have become such a common and varied software engineering exercise that much commercial and academic research has been dedicated to identifying pitfalls, techniques and component analogues for moving an application to a different target platform. The development of software or porting existing software to different, analogous system platforms have been addressed by high level languages [6,7], modular programming [9], and component based abstraction, analysis and design techniques [8]. Despite the popularity of these programming and design idioms, they generally have little support for dealing with the rather raw and limited resources available in an embedded system. Furthermore, the move towards these abstraction mechanisms has resulted in applications with footprints that do not make them feasible for deployment to a microcontroller. Though there have been efforts to port these abstraction idioms to the world of embedded systems [10], porting applications and components to a limited device configuration still require much re-engineering.

The rest of the paper is organized as follows: In section 2, we provide a brief discussion on related work in the engineering of embedded software. In section 3 background regarding network cryptography services is provided. In section 4, we provide some background information regarding the Dynamic C development environment. Section 5 discusses a number of the issues involved in porting the TLS service to the device. In section 6, we describe a few development techniques we used for our port and for embedded systems development in general. In section 7, we visit some of metrics to evaluate our port and we provide a discussion of our discoveries and insights. Finally, section 8 includes some concluding remarks.

## 2. Related Work

Cryptographic services for transport layer security (TLS) have been available as operating system and application server services for quite some time now [13]. The concept of an embedded TLS service or custom ASIC for stream ciphering have been proposed and are commercially available as SSL/TLS accelerator products from vendors such as Sun Microsystems and Cisco. They operate as black boxes and the development issues and processes to make these services available to embedded devices have been rarely discussed. Though the performance of various cryptographic algorithms such as AES and DES have been examined on many systems [19], including embedded devices [15], a discussion on the challenges of porting complete services to a device have not received such a treatment.

The scope of embedded systems development has been covered in a number of books and articles [1, 14]. Optimization techniques at the hardware design level and at the pre-processor & compiler level are well researched and benchmarked topics [1, 2, 12]. Embedded programming guidelines for optimization, style and robustness are outlined for specific languages such as ANSI C [3]. Design patterns have even been proposed in the development of embedded software to

increase portability and leverage reuse between device configurations [22].

## 3. Network Cryptographic Services

Transport Layer Security is a standard proposed by the IETF [17] and is implemented on the Internet most notably by the Secure Sockets Layer (SSL) [16]. Unfortunately, establishing and maintaining a secure connection is a computationally intensive task. Negotiating a TLS/SSL session can result in a big hit on server performance. This, in turn, can limit the number of SSL sessions which can be served and the number of concurrent SSL sessions that can be maintained at your site and it has been shown to reduce throughput by an order of magnitude [18].

The problems associated with TLS/SSL have been in part solved through processor cards that fit inside Web servers. These cards worked to offload the mathematically-intensive TLS/SSL authentication and key generation from the Web server's central CPU, and thus increase server capacity to handle more new SSL sessions. The software used to in these expansion boards is largely proprietary.

Our developing activities revolve around porting "iSSL", a minimalistic cryptographical library and network service implementation that uses the RSA and AES cipher algorithms to establish SSL-like, secure encrypted communications between two peers over the network. It includes support for session key generation and public key exchange. Due to representation and development time constraints we only port the AES cipher. Our port of the AES cipher uses the Rijndael algorithm [20], as proposed by the NIST on October 2000. The key length of AES can be independently specified to 128, 192 or 256 bits with input block lengths of 128, 192 or 256 bits. The AES cipher is required by NIST to work on a variety of processors, from 8-bit processors on smart cards and other embedded devices to powerful 64-bit workstations [19]. For our port, we referred to both the iSSL AES implementation and the AEScrypt implementation developed by Eric Green and Kaelber.

## 4. The Rabbit RMC2000 Environment

The system used for our implementation came packaged as part of an integrated board solution. It came with an integrated 10Base-T network interface and a network stack for a TCP/IP, UDP and ICMP implementation. The development board is limited to 512k flash RAM and 128k SRAM.

### 4.1 The RMC2000 Microcontroller

The Rabbit RM2000 is a high-performance 8-bit microprocessor designed to integrate with other peripherals. It has an enhanced instruction set with numerous one-byte opcodes and 16-bit logical, arithmetic, and data transfer instructions. Clock speed can be controlled via software to allow for dynamic adjustment of power and speed. It can perform 16 x 16 multiplies in 12 clock cycles and it has four levels of interrupt priorities and has 40 parallel IO lines. It has a two clock memory cycle and it has a similar architecture to that of the HDC640 and the Z180. Its 1-megabyte code space allows for C programs of up to 50,000+ lines of code. The extended Z80-style instruction set is C-friendly, with short and fast instructions for most common C operations.

This particular embedded system is designed to accommodate rapid development and porting of existing applications. The Rabbit microcontroller has a 10-pin programming port that eliminates the need for in-circuit emulators. It comes with a development environment that integrates a compiler, linker, loader, debugger and diagnostics in a single package. With the Dynamic C environment, some of the engineering costs associated with tool development and support, which has been found to account for 15% of the effort in embedded systems development [11], can be minimized and factored away from our exercise.

### 4.2 Dynamic C

Dynamic C is the language used to build applications for the Rabbit Semiconductor RMC series of microprocessors and microcontrollers. It is generally described in the context of an integrated development solution for devices, and as such it integrates many compiler, linker, loader and RTOS features into the actual programming language.

#### 4.2.1 Dynamic C, the Language

Dynamic C is an ANSI C language variant. It supports embedded assembly code and stand-alone assembly code within a source file. Its grammar and syntax are similar to that of ANSI C, but it differs from a traditional ANSI C programming system running on a PC or under UNIX in that ANSI C makes many assumptions that do not apply to embedded systems. Standard C, for example, implicitly assumes that an operating system is present and that a program starts with a clean slate, whereas embedded systems may have battery-backed memory and may retain data through power cycles. Also, Dynamic C places significant constraints on how a program is structured.

The numerous include files found in typical C programs are not used because Dynamic C has a library system that automatically provides function prototypes and similar header information to the compiler before the user's program is compiled. The C "#include" directive is supplanted by a “#use” directive and components are forced to be designed as libraries and not modules. Furthermore, Dynamic C does not support the “#pragma” preprocessor directive.

Dynamic C limits memory addressing operations to support only 20-bit addresses. It relies heavily on the existence of an on-chip memory management unit (MMU) to segment memory and translates 16-bit addresses to 20-bit memory addresses. It has "shared" and "protected" keywords that help protect data shared between different contexts or stored in battery-backed memory. Furthermore, it has a set of features that allow the programmer to make fullest use of extended memory. Normally, Dynamic C takes care of memory management, but it has keywords and directives to help put code and data in the proper place. For example, the keyword "root" selects root memory (addresses within the 64K physical address space), and the keyword "xmem" selects extended memory, which means anywhere in the 1024K code space.

Dynamic C places further constraints in the semantics of the language. Bit fields and enumerated types are not supported. There are also minor differences involving extern and register keywords. The default storage class for variables is static, and not auto. Variables that are explicitly initialized in a declaration are stored in flash memory and cannot be changed.

#### 4.2.2 Dynamic C and RTOS Integration

The RM2000 utilizes a real time operating system called µC/OS-II. The µC/OS-II RTOS is a simple real-time operating system that runs on the Rabbit microprocessor and is fully integrated into the Dynamic C language and environment. The µC/OS-II RTOS is compiled into the program and loaded into the microcontroller using some of the RM2000 bootstrapping features. Unlike ANSI C, Dynamic C integrates RTOS semantics into its language constructs. Dynamic C has a construct called "function chaining." Function chaining is program structuring construct that allows special segments of code to be embedded within one or more functions. When a named function chain executes, all the segments belonging to that chain execute. Function chains allow software to perform initialization, data recovery, or other kinds of tasks on request.

Multi-tasking is natively supported in the language. The costatement construct simplifies the implementation of state machines and it allows for concurrent parallel processes to be simulated in a single program. Costatements can be voluntarily suspended and later resumed. Cofunctions are a similar construct. Cofunctions, like costatements, are used to implement cooperative multitasking. But, unlike costatements, they have a form similar to functions and they are designed to facilitate callbacks and other signaling patterns in a real-time application. Finally, Dynamic C provides the "slice" statement construct to allow the programmer to run a block of code for a specific amount of time. These constructs are supplemented with "yield", "abort", "waitfor" and "waitfordone" keywords to control scheduling, timing and event handling of these task abstractions.

## 5. Porting and Development Issues

A number of key development issues arise in determining whether or not an embedded systems development effort is going to be feasible. These considerations range from analysis and design constraints to development tool support for supporting the target device.

### 5.1 To Port or Not to Port

The most important thing to consider before porting any application or component is to decide whether something really should or shouldn't be ported. The key driver in this assessment is determining the reducibility of the system and the percentage of the design and code that will need to change. Embedded systems are fairly sparse in terms of the available services that it can provide, so a key factor is whether or not many of the system level services that are going to be required are going to have to be redesigned and re-engineered. Whereas standard applications use COTS solutions to leverage reuse possibilities, the use of a COTS solution can become a detriment in embedded software development. Most of the features provided by COTS solutions will need to be reanalyzed, re-engineered and re-implemented from scratch to suit the operating environment of the target embedded platform.

Unfortunately, the iSSL package is a well developed system daemon and as such it utilizes a number of UNIX specific services and a number of COTS libraries. It utilizes the GNU MPI library for performing data calculations and manipulations. The MPI library contains architecture specific code for a number of target platforms. It makes available a number of optimized assembly routines for memory

addressing and shifting operations. It makes use of a "bignum" library to provide itself with a data representation mechanism adequate enough to store and manipulate up to 1024bit keys for the RSA algorithm. The network service component used by the iSSL service is mainly a simple control flow application designed to demonstrate networked stream ciphering. Though simple enough for our embedded application, it nevertheless utilizes threads for servicing multiple requests and its socket constructs adhere to the UNIX BSD model. Since it is designed to be used in a multi-threaded environment, the library code is reentrant and can handle multiple cryptographic operations. The iSSL package also contains a good amount of dynamic memory allocation code.

The iSSL package was designed to be portable across a number of platforms. It supports Windows and many, many flavors of UNIX. Retaining this property in an embedded development context is quite impossible. Though many microcontrollers share similar, if not the same instruction sets, as in the case of the Rabbit RM2000 and the Zilog Z180 processors, software cannot be guaranteed to be portable because microcontrollers are configured with various types of other board level controllers and chip sets. As such each embedded system platform introduces its own set of nuances and variations to the operating environment. Furthermore, depending on the RTOS installed, an embedded application may or may not have to perform its own resource and device management functions. Therefore, the feasibility of an application's design is dictated by the platform's architecture and configuration. Domain specific designs such as ASIPs, ASICs and tools such as SDLs, have forced embedded software to be re-engineered from scratch almost every time. It has been shown that software development for embedded applications and firmware account for up to 33% of the manpower allocation for many embedded systems projects [11].

### 5.2 Factoring an Application for an Embedded Environment

It was quite evident that porting this TLS service would require a good amount of rework. Aside from the obvious platform constraints, much of the logical code would remain the same. Much of the work involved was with re-architecting certain portions of the system to fit within an embedded context. A certain amount of reduction had to take place to simplify operations and task handling and to re-engineer the interfaces to the network and the data. The control flow of the application, task scheduling and error handling all became issues in this new context. Furthermore, the largely user delegated error recovery and error handling semantics of the TLS service had to be re-designed to accommodate the possibility of continuous operation in a real time, autonomous setting. Since a file semantics did not exist in the controller, logging mechanisms for error reporting had to be either discarded or re-engineered as part of the protocol. Platform abstractions needed to be refactored or eliminated to minimize code size. Extra interfaces and abstractions add to the size of the application image, especially when embedded software tools do not support dynamic linkers and loaders. Usually, as in the case of the RMC2000, all code is statically linked and a single image is loaded to the EPROM.

### 5.3 Resource Management

Maintaining state within the application was a big design issue. Memory is a precious resource in a microcontroller and the way Dynamic C statically allocates variables made it difficult to port the current book-keeping logic. The original version of iSSL makes use of automatic variables and temporaries and state is saved in per thread stacks for reentrancy.

Without a full featured operating system to manage low level resources a number of low level issues had to be considered. Issues regarding how the processor saves its state had to be examined to determine whether or not register values had to be manually saved or restored before returning from an interrupt. Register preservation require multi-byte transfer routines which can add up for often used/called interrupt. The frequency of the interrupt calls and the amount of work to be done within them needed to be estimated.

Since code and data space were quite limited, the size of the compiled code would have to be constantly monitored. During the development process, code size changes depending on whether the compilation mode is designed to deliver a debug, non-optimized or optimized version of the application image or to include or exclude a specific language or application feature.

## 6. Development Techniques

Embedded systems are application specific and they are not usually reprogrammed or upgraded during their lifetimes. As such, the design can be tuned for very specific functions. Furthermore, an embedded system is reactive. It reacts to events from the environment and performs certain kinds of processing under some real time constraint. In contrast to general purpose systems, where a goal is to maximize performance, with a reactive embedded system, the performance is a constraint. We discuss below a number of strategies

that are available to allow a piece of embedded software to perform within such tight constraints.

### 6.1 Analysis and Design Strategies

During analysis and design of this port, resources had to be scoped and a strategy for its allocation and use had to be determined ahead of time. One of the first activities that had to be accommodated was interrupt planning [3]. A number of ISR handling strategies are available [3, 4] and our interrupt handling strategy involved deciding ahead of time whether or not certain processing tasks were going to be part of the interrupt service routine or part of the main-line code. We analyzed the reentrancy each code block supported in the original system and we designed the type of reentrancy we were allowed to support in the microcontroller. This involved considering the number of temporaries we needed to use and analyzed how much work was involved in saving and restoring processor state. Dynamic C's ISR facilities made it easy to partition these activities, but its default static allocation strategy meant breaking a number of reentrancy programming rules [1, 3, 4]. It has well defined semantics for how the main-line code and the ISR interact, but the details behind saving and restoring specific registers would make it much harder to debug and test. Given that these ISRs are not reentrant and that the frequency of context switching between servicing network connections and processing the data, we had decided on placing a majority of the handler code within the main-line section of the program. Furthermore, it would have been more difficult to handle errors and state changes within the application when non-maskable interrupts (NMI) needed to be processed. It would be difficult to estimate the debugging and testing time required from the resulting increase in software complexity. Simplicity dominated our design strategy for many such related issues.

Transitioning to the idea of ROM'd code, where the data and code is stored separately, required a change in design and programming mindset. We had to carefully plan how much memory the program would use and how much code space the compiled application image would take on the device. We used a well defined taxonomy [4] for considering the memory footprint of certain software components that were going to be compiled into the image. Calculating code space requirements was easy compared to determining the run-time size requirements of the application. The Dynamic C environment provided tools for monitoring code space and the application's image footprint. Memory pre-allocation and planning were used for most of the complex function points in the system. This type of analysis was a bit more difficult. Key sizes and block input sizes had to be constrained in order to make memory usage deterministic. The original AES library implementation supplied with iSSL supported various key and block sizes. Porting this application required that we go with a fixed key and input block size of 128bits, occupying a fixed size of 32 bytes. It is a good compromise for maintaining the cryptographic strength of the application while limiting the application enough to make development deterministic and manageable. Furthermore, the complexity involved in developing a "BigNum" library made it unfeasible to port the RSA algorithm.

Organizing the system and its control flow is another challenge in embedded systems design. There are various system organizations within real-time systems that were considered for use [21]. From the initial analysis stage it was clear that the application port could only be organized one of three ways as outlined by Perkshot's real-time systems taxonomy [21]. In order to maximize throughput, we considered a level 0 organization, called a polling loop. This system exhibits no complexity attributes and is a single loop with sequential execution and almost no interrupts. Branches are allowed to select alternative actions depending upon input values. A single timer interrupt is sometimes allowed as long as it is used only to update a clock or time variable. The polling loop executes continuously and endlessly, and the loop time is determined only by the amount of work performed for every iteration. A level 1 schedule loop was also considered for use. It does not visit a fixed list of inputs and tasks, but instead continuously polls a dynamic schedule [21]. Tasks are placed on the schedule either repetitively, by themselves, or by other, previously executed tasks or functions. With further analysis, a level 2, preemptive single task strategy was decided as the organization of choice for our application. This organization is a bit more complex, but the event handling semantics that we needed support made it time consuming to retrofit to the simpler level 0 and level 1 strategies. The Dynamic C language and the RTOS supplied by the RMC2000 natively supported both timing and priority complexity attributes required to perform stateful network and cryptographic operations.

Design patterns are normally associated with complex, object oriented application services. The iSSL network service, for example, utilizes a complex pre-threading/pre-forking strategy to service incoming connections. Though not normally associated with embedded systems, we have found a number of applicable design patterns to apply to our system. In our specific embedded application domain, we have found certain design patterns to encapsulate task

abstractions well and to conceptually decouple subsystems. Dynamic C provides a number of primitive constructs that were helpful in providing task abstractions. The function chaining construct and the costatement/cofunction construct allowed for ISRs and tasks to be designed in a decoupled fashion. A timer pattern was used to abstract clock timing functions from tasks [22]. The Dynamic C slice construct facilitated the ease at which we were able to utilize this construct since it integrated the concept of a real-time clock handler and scheduler into one. Costatement tasks could then encapsulate critical sections and be run in these "slices" from a very simple scheduler proper. With such constructs supported in the language, abstractions to support scheduling policies can be used. Using the macro expansion of Dynamic C, a family of scheduler policies can be parameterized with function blocks to slice costatements.

### 6.2 Programming Idioms for Embedded Systems

The correlation between code size and performance is an openly debated issue with all systems. The story is a bit more straightforward with embedded systems. The DSPStone project [12] showed that code size was directly related to performance overhead. In some embedded systems, memory is integrated into the CPU core, and thus a larger code size immediately implies a lower chip yield and higher costs. Optimization can have a both a positive effect on code size and performance, while in other cases, there is a trade off between the two optimization goals. Eventually, it depends on the concrete application and technology at hand, which optimization goal should be priority [2]. For our system, performance and accurate data representation are key factors. In order to maximize performance with the proper key representations, a number of techniques were used for both the control and library functions.

One general rule that had to be followed was to write global variables in one place only. In order to maximize efficiency in the face of frequent context switching for interrupt handlers, state data either had to be in the main line or interrupt code. Data exchange should only be accomplished by read them into the other. The key to making this possible was ensuring that the communication between the interrupt routine and the main-line code travel one way if possible [4].

One piece of device knowledge that the programmer must keep in mind is the endianness of the processor. C does not deal with endianness even in multi-byte shift operations and quirks are known to appear in microcontrollers. We had to ensure that byte orderings were little-endian. Fortunately, the network routines that come with the Rabbit development environment take care of bit packing and converting byte orderings.

Unrolling for loops have been documented to save ROM space [4] and was considered for this port, but this technique was not used for our application port. The Dynamic C compiler is a black box and determining which code optimizations it used was impossible. Hand optimization features such as loop unrolling may be obsoleted by the compiler. The benefits of implement such a feature did not warrant the time and effort it would have required to test and debug and verify the application.

We made good use of the macro facility provided by the compiler. Software tools sometimes lag behind new processors and it is not uncommon to use a derivative CPU with an instruction that has not been implemented in the assembler [1]. Macros are generally used to abstract hardware nuances and place new instructions within a section of code without having to redesign the program's structure. For our application, though, since the language couples our software with the Rabbit RMC2000 platform, we used macros to facilitate simple design patterns.

## 7. Performance and Detailed Discoveries

Our port utilizes a naïve and an optimized version of the AES library to see which optimization techniques affect performance. The optimized AES port is a pure assembly implementation derived from the work of Paulo S.M. Barreto. This optimized version sacrifices memory for speed. The data representation is complex to represent and costly to use. 32 bit integer types are represented by two segments of 16bit unsigned integers to represent the high and low order bits of the number.

Development time is dominated by the steep learning curve to learn the Dynamic C constructs and the RMC2000 platform. Small language features, constructs or design changes take a non-trivial amount of time to implement, debug and test. Most of the engineering effort was originally thought to be tied up in rework and reconfiguration. The actual effort allocation was dominated by debugging and testing. There were some unexpected and undocumented hardware behaviors and the semantics of Dynamic C language seem to run contrary to that of the documented specifications.

| Development Time | | |
|---|---|---|
| | Actual(PM) | Estimated(PM) |
| AES | 0.2 | 0.1 |
| AES Assembly | 0.3 | 0.1 |
| Application | 0.6 | 0.3 |

From our experience, all we can conclude is that having an eye towards redesigning from scratch is better. Even library routines, which are usually meant to be portable, cannot be reused if performance is a critical factor. Designing for portability only serves to reduce the cost of deploying certain libraries across other device platforms. Certain design patterns help, but are by no means meant to improve performance. They merely serve to make conceptual designs clear in code and do not translate directly to reduce size or to increase portability and performance.

| TLS w/AES | | | | |
|---|---|---|---|---|
| ***Compilation options*** | ***1,2,3*** | ***3*** | ***1,2,4*** | ***4*** |
| SLOC | 7532 | 7365 | 7513 | 7495 |
| code size (bytes) | 28575 | 27129 | 28043 | 26895 |
| encrypt time (secs) | 0.064 | 0.054 | 0.064 | 0.059 |
| decrypt time (secs) | 0.082 | 0.076 | 0.097 | 0.077 |
| **TLS w/AES assembly** | | | | |
| ***Compilation options*** | ***1,2,3*** | ***3*** | ***1,2,4*** | ***4*** |
| SLOC | 7259 | 7294 | 7250 | 7213 |
| code size (bytes) | 24325 | 24139 | 23794 | 23463 |
| encrypt time (secs) | 0.0022 | 0.0022 | 0.0022 | 0.0022 |
| decrypt time (secs) | 0.0031 | 0.0031 | 0.0031 | 0.0031 |
| **compilation options** | Type checking 1 | Debug symbols 2 | Speed optimization 3 | Size reduction 4 |

The assembly version of the TLS service performed faster than our design by a factor of 15-20. Hand optimizations, creative use of dynamic C constructs and compiler optimizations failed to bring our design within a reasonable distance from the hand-coded assembly version. Good design in an embedded world is hard to define since the metrics differ from that of a standard application. Requirements satisfaction and real-time operation are more important than portability, scalability and elegance. Algorithm optimization beats out most optimizations performed in the software architecture level. From the data we have collected, there seems to be little correlation between code size and performance. An 8% difference in SLOC and a 9% difference in binary size translated to an order of magnitude speed difference.

We have devised a series of "simplifiers and complicators" [23] that can be used as a rough guideline to assess the relative amount of software and development complexity that might be encountered in an embedded software project.

| simplifiers | complicators |
|---|---|
| HLL support | No language support |
| Integrated tool environment | Non standard algorithm support |
| Optimized library support | Fault tolerance and recovery |
| Memory availability | Real-time performance |
| Platform maturity | Multi-tasking |
| Soft real-time performance | IPC requirements |
| Stateless application | Memory intesiveness |

## 8. Conclusion

Software development on an embedded platform requires a different mindset. The benchmarks for success, performance, portability run contrary to that of standard application or systems development. The interplay between performance, code size and memory footprint determine the success of any embedded systems project. This usually translates to a development process dominated by planning and testing and verification. Design elegance and peephole optimizations provide spotty performance gains and are not favored over algorithmic optimizations and instruction set intimacy.

## References


1. J. G. Gassle, The Art of Programming Embedded Systems. Academic Press Inc, 1992
2. R. Leupers, Code Optimization Techniques for Embedded Processors: methods, Algorithms, and Tools. Kulwer Academic Publishers, 2000
3. M. Barr, Embedded System Programming in C and C++. O'Reilly & Associates, 1999
4. K. Zurell, C Programming for Embedded Systems. CMP Books, 2000.
5. P. Koopman, "Embedded System Design Issues -- The Rest of the Story", Proceedings of the 1996 International Conference on Computer Design, Austin, October 7-9 1996.
6. P. J. Brown, "Levels of language for portable software," Communications of the ACM 15(12), pg 1059-1062, Dec. 1972.
7. B.W. Kernigham and D. M. Ritchie, "The C Programming (ANSI C) Language", 2nd. Edition, Prentice Hall, Englewood Cliffs, New Jersey
8. S. Vinoski, "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments," IEEE

Communications Magazine, vol. 14, no. 2, February 1997.
9. D.R. Hanson, C Interfaces and Implementations-Techniques for Creating Reusable Software. Reading, MA: Addison-Wesley, 1997.
10. A. Gokhale and D. C. Schmidt, "Techniques for Optimizing CORBA Middleware for Distributed Embedded Systems," in Proceedings of INFOCOM '99, Mar. 1999.
11. P. G.Paulin, et.al., "Trends in Embedded Systems Technology: An Industrial Perspective", NATO ASI on Hardware/Software Codesign, Lake Como, Italy, 1995.
12. V. Zivojnovic. C. Schlager, H. Meyr, "DSPStone: A DSP Oriented Benchmarking Methodology," International Conference on Signal Processing, 1995.
13. Ralf S. Engelschall. mod SSL, 2000. WWW documentation. (better-documented derivative of the Apache SSL secure web server) http://www.modssl.org
14. J. Gassle, "Dumb Mistakes", The Embedded Muse New Letter, August 7, 1997.
15. C. Yang, "Performance Evaluation of AES/DES/Camellia on the 6805 and H8/300 CPUs", In the Proceedings of the 2001 Symposium on Cryptography and Information Security," pgs. 727-730, Oiso, Japan, January 23, 2001.
16. A. O. Freier, P. Karlton, and P. C. Kocher, "The SSL protocol," Internet draft, Transport Layer Security Working Group, Nov. 1996.
17. T. Dierks and C. Allen, "The TLS protocol," Internet draft, Transport Layer Security Working Group, May 1997.
18. Arthur Goldberg, Robert Buff, Andrew Schmitt, "Secure Web Server Performance using SSL Session Keys", Published in the "Workshop on Internet Server Performance", held in conjunction with SIGMETRICS'98, June 23, 1998.
19. B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson, "Performance Comparison of the AES Submissions," Version www.counterpane.com, January 1999.
20. J. Daemen and V. Rijmen. "The block cipher Rijndael," In Third Smard Card Research and Advanced Applications Conference Proceedings, 1998.
21. G.G. Preckshot. "Real-Time Systems Complexity and Scalability," In a FEESP Technical Report for Lawrence Livermore National Laboratory, May 28, 1993.
22. M. de Champlain. "Patterns to Ease the Port of Micro-kernels in Embedded Systems," In Proc. of the 3rd Annual Conference on Pattern Languages of Programs (PLoP'96), Allerton Park, IL, June 1996.
23. B. Boehm, D. Port, A. Egyed "The MBASE Life Cycle Architecture Milestone Package: No Architecture Is An Island," *1st Working International Conference on Software Architecture*, 1999.