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

Stephen Jan Paolo de Dios Stephen A. Edwards
Department of Computer Science, Columbia University
1214 Amsterdam Avenue, New York, New York, 10027
{sj178, pd119}@columbia.edu sedwards@cs.columbia.edu

## Abstract

*This paper describes our experience porting a transport-layer 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 chose for this case study. We examine the effectiveness of a few proposed porting strategies by examining important program and run-time characteristics.*

### 1 Introduction

Embedded systems present a different software engineering problem. These systems are unique in that the hardware and the software are tightly integrated. The limited nature of an embedded systems operating environment requires a different approach to developing and porting software. In this paper, we discuss the key issues in developing and porting a Unix system-level transport-level security (TLS) service to an embedded microcontroller. We discuss our design decisions and experience porting this service using Dynamic C, a C variant, on the RMC2000 microcontroller from Rabbit Semiconductor[1]. The main challenges came when APIs for operating-system services such as networking were either substantially different or simply absent.

Porting software across platforms is 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 it. Porting software has been addressed by high level languages [2, 12], modular programming [11], and component based abstraction, analysis and design techniques [17]. Despite the popularity of these techniques, they are of limited use when dealing with the limited and rather raw resources of a typical embedded system. In fact, these abstraction mechanisms tend to consume more resources, especially memory, making them impractical for microcontrollers. Though some have tried to migrate some of these abstractions to the world of embedded systems [9], porting applications in a resource-constrained system still requires much reengineering.

This paper presents our experiences porting a small networking service to an embedded microcontroller with an eye toward illustrating what the main problems actually are. Section 2 introduces the network cryptographic service we ported. Section 3 describes some relevant related work, and Section 4 describes the target of our porting efforts, the RMC 2000 development board.

Section 5 describes issues we encountered while porting the cryptographic network service to the development board, Section 6 describes the performance experiments we conducted; we summarize our findings in Section 7.

### 2 Network cryptographic services

For our case study, we ported iSSL,[2] a public-domain implementation of the Secure Sockets Layer (SSL) protocol [6], a Transport-Layer Security (TLS) standard proposed by the IETF [5]. SSL is a protocol that layers on top of TCP/IP to provide secure communications, e.g., to encrypt web pages with sensitive information.

Security, sadly, is not cheap. Establishing and maintaining a secure connection is a computationally-intensive task; negotiating an SSL session can degrade server performance. Goldberg et al. [10] observed SSL reducing throughput by an order of magnitude.

iSSL is a cryptographic library that layers on top of the Unix sockets layer to provide secure point-to-point communications. After a normal unencrypted socket is created, the iSSL API allows a user to bind to the socket and then do secure read/writes on it.

To gain experience using the library, we first implemented a simple Unix service that used the iSSL library to establish a secure redirector. Later, we ported this service to the RMC2000.

[1] http://www.rabbitsemiconductor.com

[2] http://sourceforge.net/projects/issl

Because SSL forms a layer above TCP, it is easily moved from the server to other hardware. For performance, many commercial systems use coprocessor cards that perform SSL functions. Our case study implements such a service.

The iSSL package uses the RSA and AES cipher algorithms and can generate session keys and exchange public keys. Because the RSA algorithm uses a difficult-to-port bignum package, we only ported the AES cipher, which uses the Rijndael algorithm [3]. By default, iSSL supports key lengths of 128, 192, or 256 bits and block lengths of 128, 192, and 256 bits, but to keep our implementation simple, we only implemented 128-bit keys and blocks. During porting, we also referred to the AESCrypt implementation developed by Eric Green and Randy Kaelber[3].

## 3 Related work

Cryptographic services for transport layer security (TLS) have long been available as operating system and application server services [15]. The concept of an embedded TLS service or custom ASIC for stream ciphering 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 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 [16], including embedded devices [18], 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 [7, 8]. Optimization techniques at the hardware design level and at the pre-processor and compiler level are well-researched and benchmarked topics [8, 14, 19]. Guidelines for optimizing and improving the style and robustness of embedded programs have been proposed for specific languages such as ANSI C [1]. Design patterns have also been proposed to increase portability and leverage reuse among device configurations for embedded software [4].

Overall, we found the issues involved in porting software to the embedded world have not been written about extensively, and are largely considered "just engineering" doomed to be periodically reinvented. Our hope is that this paper will help engineers be more prepared in the future.

## 4 The RMC2000 environment

Typical for a small embedded system, the RMC2000 TCP/IP Development Kit includes 512k of flash RAM, 128k SRAM, and runs a 30 MHz, 8-bit Z80-based microcontroller (a Rabbit 2000). While the Rabbit 2000, like the Z80, manipulates 16-bit addresses, it can access up to 1 MB through bank switching.

[3] http://aescrypt.sourceforge.net

The kit also includes a 10Base-T network interface and comes with software implementing TCP/IP, UDP and ICMP. The development environment includes compilers and diagnostic tools, and the board has a 10-pin programming port to interface with the development environment.

### *4.1 Dynamic C*

The Dynamic C language, developed along with the Rabbit microcontrollers, is an ANSI C variant with extensions that support the Rabbit 2000 in embedded system applications. For example, the language supports cooperative and preemptive multitasking, battery-backed variables, and atomicity guarantees for shared multibyte variables.

Unlike ANSI C, local variables in Dynamic C are `static` by default. This can dramatically change program behavior, although it can be overridden by a directive.

Dynamic C does not support the `#include` directive, using instead `#use`, which gathers precompiled function prototypes from libraries. Deciding which `#use` directives should replace the many `#include` directives in the source files took some effort.

Dynamic C omits and modifies some ANSI C behavior. Bit fields and enumerated types are not supported. There are also minor differences in the `extern` and `register` keywords. As mentioned earlier, the default storage class for variables is `static`, not `auto`, which can dramatically change the behavior of recursively-called functions. Variables initialized in a declaration are stored in flash memory and cannot be changed.

Dynamic C's support for inline assembly is more comprehensive than most C implementations, and it can also integrate C into assembly code, as in the following:

```
#asm nodebug
InitValues::
   ld hl,0xa0;
c  start_time = 0;   // Inline C
c  counter = 256;    // Inline C
   ret
#endasm
```

We used the inline assembly feature in the error handling routines that caught exceptions thrown by the hardware or libraries, such as divide-by-zero. We could not rely on an operating system to handle these errors, so instead we specified an error handler using the `defineErrorHandler(void *errfcn)` system call. Whenever the system encounters an error, the hardware passes information about the source and type of error on the stack and calls this user-defined error handler. In our implementation, we used (simple) inline assembly statements to retrieve this information. Because our application was not designed for high reliability, we simply ignored most errors.

### *4.2 Multitasking in Dynamic C*

Dynamic C provides both cooperative multitasking, through costatements and cofunctions, and preemptive multitasking through either the `slice` statement or a port of Labrosse's µC/OS-II real-time operating system [13].

Dynamic C's costatements provide multiple threads of control through independent program counters that may be switched among explicitly, such as in this example:

```
for (;;) {
  costate {
    waitfor( tcp_packet_port_21() );
    // handle FTP connection
    yield; // Force context switch
  }
  costate {
    waitfor( tcp_packet_port_23() );
    // handle telnet connection
  }
}
```

The `yield` statement immediately passes control to another costatement. When control returns to the costatement that has yielded, it resumes at the statement following the `yield`. The statement `waitfor(`*`expr`*`)`, which provides a convenient mechanism for waiting for a condition to hold, is equivalent to `while (!`*`expr`*`) yield;`.

Cofunctions are similar, but also take arguments and may return a result.

In our port, we used costatements to handle multiple connections with multiple processes. We did not use µC/OS-II.

### *4.3 Storage class specifiers*

To avoid certain race conditions, Dynamic C generates code that disables interrupts while multibyte variables marked `shared` are being changed, guaranteeing atomic updates.

For variables marked `protected`, Dynamic C generates extra code that copies their value to battery-backed RAM before every modification. Backup values are copied to main memory when when system is restarted or when `_sysIsSoftReset()` is called. We did not need this feature in this port.

The Rabbit 2000 microcontroller has a 64K address space but uses bank-switching to access 1M of total memory. The lower 50K is fixed, "root" memory, the middle 6K is I/O, and the top 8K is bank-switched access to the remaining memory. A user can explicitly request a function to be located in either root or extended memory using the storage class specifiers `root` and `xmem` (Figure 1).

We explicitly located certain functions, such as the error handler, in root memory, but we let the compiler locate the others.

```
// Interrupts disabled during changes to a, b, and c
// Updates guaranteed atomic
shared float a, b, c;

main() {
   protected int state1; // Battery-backed
   ...
   // restore protected variables
   _sysIfSoftReset()
}

// Place func1 in root memory
root int func1() { ... }

// Place following assembly code in root memory
#memmap root
#asm root
 ...
#endasm

// Place func2 in extended memory
xmem int func2() { ... }
```

**Figure 1. Fragment illustrating various Dynamic-C-specific storage class specifiers.**

### *4.4 Function chaining*

Dynamic C provides function chaining, which allows segments of code to be embedded within one or more functions. Invoking a named function chain causes all the segments belonging to that chain to execute. Such chains enable initialization, data recovery, or other kinds of tasks on request. Our port did not use this feature.

```
// Create a chain named ''recover'' and add three functions
#makechain recover
#funcchain recover free_memory
#funcchain recover declare_memory
#funcchain recover initialize

// Invoke all three functions in the chain in some sequence
recover();
```

## 5 Porting and development issues

A program rarely runs unchanged on a dramatically different platform; something always has to change. The fundamental question is, then, how much must be changed or rewritten, and how difficult these rewrites will be.

We encountered three broad classes of porting problems that demanded code rewrites. The first, and most common, was the absence of certain libraries and operating system facilities. This ranged from fairly simple (e.g., Dynamic C does not provide the standard `random` function),

```
int echo_server() {
  int sock, newsock, len;
  struct sockaddr_in addr;
  char buf[LEN];

  if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    return -1;

  memset(&addr, 0, sizeof(addr));
  addr.sin_family      = AF_INET;
  addr.sin_addr.s_addr = htonl(INADDR_ANY);
  addr.sin_port        = htons(MYPORT);
  if ( bind(sock, (struct sockaddr *) &addr,
            sizeof(struct sockaddr_in)) < 0 ) return -1;
  if ( listen(sock, LISTENQ) < 0 ) return -1;
  for (;;) {
    if ((newsock = accept(sock, NULL, NULL) ) < 0 )
      return -1;
    if ((len = recv(newsock, buf, LEN, 0)) < 0)
      return -1;
    if (send(newsock, buf, len, 0) < 0) return -1;
    close(conn_s);
  }
}
```

(a)

```
int echo_server()
{
  tcp_Socket sock;
  int status;
  char buf[LEN];

  sock_init();
  for (;;) {
    tcp_listen(&sock, PORT, 0, 0, NULL, 0);
    sock_wait_established(&sock, 0, NULL, &status);
    sock_mode(&sock, TCP_MODE_ASCII);
    while (tcp_tick(&sock)) {
      sock_wait_input(&sock, 0, NULL, &status);
      if (sock_gets(&sock, buf, LEN))
        sock_puts(&sock, buf);
    }
  }
}
```

(b)

**Figure 2. A comparison of (a) traditional BSD sockets-based code and (b) equivalent code in the Dynamic C environment illustrating the significant differences in API.**

to fairly difficult (e.g., the protocols include timeouts, but Dynamic C does not have a timer), to virtually impossible (e.g., the iSSL library makes some use of a filesystem, something not provided by the RMC2000 environment). Our solutions to these ranged from creating a new implementation of the library function (e.g., writing a `random` function) to working around the problem (e.g., changing the program logic so it no longer read a hash value from a file) to abandoning functionality altogether (e.g., our final port did not implement the RSA cipher because it relied on a fairly complex bignum library that we considered too complicated to rework).

A second class of problem stemmed from differing APIs with similar functionality. For example, the protocol for accessing the RMC2000's TCP/IP stack differs quite a bit from the BSD sockets used within iSSL. Figure 2 illustrates some of these differences. While solving such problems is generally much easier than, say, porting a whole library, reworking the code is tedious.

A third class of problem required the most thought. Often, fundamental assumptions made in code designed to run on workstations or servers, such as the existence of a filesystem with nearly unlimited capacity (e.g., for keeping a log), are impractical in an embedded systems. Logging and somewhat sloppy memory management that assumes the program will be restarted occasionally to cure memory leaks are examples of this. The solutions to such problems are either to remove the offending functionality at the expense of features (e.g., remove logging altogether), or a serious reworking of the code (e.g., to make logging write to a circular buffer rather than a file).

### *5.1 Interrupts*

We used the serial port on the RMC2000 board for debugging. We configured the serial interface to interrupt the processor when a character arrived. In response, the system either replied with a status messages or reset the application, possibly maintaining program state.

A Unix environment provides a high-level mechanism for handling software interrupts:

```
main() {
  signal(SIGINT, sigproc); // Register signal handler
}
void sigproc() { /* Handle the signal */ }
```

In Dynamic C, we had to handle the details ourselves. For example, to set up the interrupt from the serial port, we had to enable interrupts from the serial port, register the interrupt routine, and enable the interrupt receiver.

```
main() {
  // Set serial port A as input interrupt
  WrPortI(SADR, &SADRShadow, 0x00);
  // Register interrupt service routine
  SetVectExtern2000(1, my_isr);
  // Enable external INT0 on SA4, rising edge
  WrPortI(I0CR, NULL, 0x2B);
  ⋮
  // Disable interrupt 0
  WrPortI(I0CR, NULL, 0x00);
}

nodebug root interrupt void my_isr() { ... }
```

We could have avoided interrupts had we used another network connection for debugging, but this would have made it impossible to debug a system having network communication problems.

### *5.2 Memory*

A significant difference between general platform development and embedded system development is memory. Most embedded devices have little memory compared to a typical modern workstation. Expecting to run into memory issues, we used a well-defined taxonomy [20] to plan out memory requirements. This proved unnecessary, however, because out application had very modest memory requirements.

Dynamic C does not support the standard library functions `malloc` and `free`. Instead, it provides the `xalloc` function that allocates extended memory only (arithmetic, therefore, cannot be performed on the returned pointer). More seriously, there is no analogue to `free`; allocated memory cannot be returned to a pool.

Instead of implementing our own memory management system (which would have been awkward given the Rabbit's bank-switched memory map), we chose to remove all references to `malloc` and statically allocate all variables. This prompted us to drop support of multiple key and block sizes in the iSSL library.

### *5.3 Program structure*

As we often found during the porting process, the original implementation made use of high-level operating system functions such as `fork` that were not provided by the RMC2000 environment. This forced us to restructure the program significantly.

The original TLS implementation handles an arbitrary number of connections using the typical BSD sockets approach shown below. It first calls `listen` to begin listening for incoming connections, then calls `accept` to wait for a new incoming connection. Each request returns a new file descriptor passed to a newly-`fork`ed process that handles the request. Meanwhile, the main loop immediately calls `accept` to get the next request.

```
listen(listen_fd)
for (;;) {
  accept_fd = accept(listen_fd);
  if ((childpid = fork()) == 0) {
     // process request on accept_fd
     exit(0); // terminate process
  }
}
```

The Dynamic C environment provides neither the standard Unix `fork` nor an equivalent of `accept`. In the RMC 2000's TCP implementation, the socket bound to the port also handles the request, so each connection is required to have a corresponding call to `tcp_listen`. Furthermore,

```
for (;;) {
  costate {
    tcp_listen(socket1,TLS_PORT, ...);
    while (sock_established(socket1) == 0) yield;
    // handle request
  }
  costate {
    tcp_listen(socket2,TLS_PORT, ...);
    while((0 == sock_established(socket2))) yield;
    // handle request
  }
  costate {
    tcp_listen(socket2,TLS_PORT, ...);
    while((0 == sock_established(socket2))) yield;
    // handle request
  }
  costate {
        // drive TCP stack
        tcp_tick(NULL);
  }
}
```

**Figure 3. The structure of the main loop of the TLS server, which can handle at most three requests because it is limited to four processes.**

Dynamic C effectively limits the number of simultaneous connections by limiting the number of costatements.

Thus, to handle multiple connections and processes, we split the application into four processes: three processes to handle requests (allowing a maximum of three connections), and one to drive the TCP stack (Figure 3). We could easily increase the number of processes (and hence simultaneous connections) by adding more costatements, but the program would have to be re-compiled.

## 6 Experimental results

To gauge which optimization techniques were worthwhile, we compared the C implementation of the AES algorithm (Rijndael) included with the iSSL library with a hand-coded assembly version supplied by Rabbit Semiconductor. A testbench that pumped keys through the two implementations of the AES cipher showed the assembly implementation ran faster than the C port by a factor of 15–20.

We tried a variety of optimizations on the C code, including moving data to root memory, unrolling loops, disabling debugging, and enabling compiler optimization, but this only improved run time by perhaps 20%.

Code size appeared uncorrelated to execution speed. The assembly implementation was 9% smaller than the C, but ran more than an order of magnitude faster.

Debugging and testing consumed the majority of the development time. Many of these problems came from our lack of experience with Dynamic C and the RMC2000 platform, but unexpected, undocumented, or simply contradictory behavior of the hardware or software and its specifications also presented challenges.

## 7 Conclusions

We described our experiences porting a library and server for transport-level security protocol—iSSL—onto a small embedded development board: the RMC 2000, based on the Z80-inspired Rabbit 2000 microcontroller. While the Dynamic C development environment supplied with the board gave useful, necessary support for some hardware idiosyncrasies (e.g., its bank-switched memory architecture) its concurrent programming model (cooperative multitasking with language-level support for costatements and cofunctions) and its API for TCP/IP both differed substantially from the Unix-like behavior the service originally used, making porting difficult.

Different or missing APIs proved to be the biggest challenge, such as the substantial difference between BSD-like sockets and the provided TCP/IP implementation or the simple absence of a filesystem. Our solutions to these problems involved either writing substantial amounts of additional code to implement the missing library functions or reworking the original code to use or simply avoid the API.

We compared the speed of our direct port of a C implementation of the RSA (Rijndael) ciper with a hand-optimized assembly version and found a disturbing factor of 15–20 in performance in favor of the assembly.

From all of this, we conclude that there must be a better way. Understanding and dealing with differences in operating environment (effectively, the API) is a tedious, error-prone task that should be automated, yet we know of no work beyond high-level language compilers that confront this problem directly.

## References


[1] M. Barr. *Programming Embedded Systems in C and C++*. O'Reilly & Associates, Inc., Sebastopol, California, 1999.

[2] P. J. Brown. Levels of language for portable software. *Communications of the ACM*, 15(12):1059–1062, Dec. 1972.

[3] J. Daemen and V. Rijmen. The block cipher Rijndael. In *Proceedings of the Third Smart Card Research and Advanced Applications Conference*, 1998.

[4] M. de Champlain. Patterns to ease the port of micro-kernels in embedded systems. In *Proceedings of the 3rd Annual Conference on Pattern Languages of Programs (PLoP'96)*, Allterton Park, Illinois, June 1996.

[5] T. Dierks and C. Allen. The TLS protocol. Internet draft, Transport Layer Security Working Group, May 1997.

[6] A. O. Freier, P. Karlton, and P. C. Kocher. The SSL protocol. Internet draft, Transport Layer Security Working Group, Nov. 1996.

[7] J. Gassle. Dumb mistakes. *The Embedded Muse Newsletter*, August 7 1997.

[8] J. G. Gassle. *The Art of Programming Embedded Systems*. Academic Press, 1992.

[9] A. Gokhale and D. C. Schmidt. Techniques for optimizing CORBA middleware for distributed embedded systems. In *Proceedings of INFOCOM '99*, Mar. 1999.

[10] A. Goldberg, R. Buff, and A. Schmitt. Secure web server performance using SSL session keys. In *Workshop on Internet Server Performance, held in conjunction with SIGMETRICS'98*, June 1998.

[11] D. R. Hanson. *C Interfaces and Implementations-Techniques for Creating Reusable Software*. Addison-Wesley, Reading, Massachussets, 1997.

[12] B. W. Kernighan and D. M. Ritchie. *The C Programming Langage*. Prentice Hall, Englewood Cliffs, New Jersey, second edition, 1988.

[13] J. Labrosse. *MicroC/OS-II*. CMP Books, Lawrence, Kansas, 1998.

[14] R. Leupers. *Code Optimization Techniques for Embedded Processors: Methods, Algorithms, and Tools*. Kluwer Academic Publishers, 2000.

[15] mod ssl. Documentation at http://www.modssl.org, 2000. Better-documented derivative of the Apache SSL secure web server.

[16] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson. Performance comparison of the AES submissions. In *Proceedings of the Second AES Candidate Conference*, pages 15–34, NIST, Mar. 1999.

[17] S. Vinoski. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), Feb. 1997.

[18] C. Yang. Performance evaluation of AES/DES/Camellia on the 6805 and H8/300 CPUs. In *Proceedings of the 2001 Symposium on Cryptography and Information Security*, pages 727–730, Oiso, Japan, Jan. 2001.

[19] V. Zivojnovic, C. Schlager, and H. Meyr. DSPStone: A DSP-oriented benchmarking methodology. In *International Conference on Signal Processing*, 1995.

[20] K. Zurell. *C Programming for Embedded Systems*. CMP Books, 2000.