Secure Sockets Layer (SSL) Protocol

1.0 Introduction

What is SSL?

SSL is the ubiquitous security protocol used in almost 100% of secure Internet transactions. Essentially, SSL transforms a typical reliable transport protocol (such as TCP) into a secure communications channel suitable for conducting sensitive transactions.[i] The SSL protocol defines the methods by which a secure communications channel can be established—it does not indicate which cryptographic algorithms need to be used. SSL supports many different algorithms, and serves as a framework whereby cryptography can be used in a convenient and distributed manner.

Uses for SSL

The uses for SSL are endless. Any application that needs to transmit data over an unsecured network such as the Internet or a company intranet is a potential candidate for SSL. SSL provides security, and more importantly, peace of mind. When using SSL, you can be fairly sure that your data are safe from eavesdroppers and tampering.

SSL is relatively new to the embedded world because it has been too complex for traditional embedded-systems microprocessors to handle. However, starting with Rev. A of the Rabbit 3000 microprocessor, hardware assistance has been added to speed up some of the more complex SSL cryptography operations, making SSL a viable solution in a market where standard (usually complex) security protocols have not traditionally been supported. The applications for embedded applications are as numerous as those for the PC world. The following are just a few potential applications for embedded SSL.

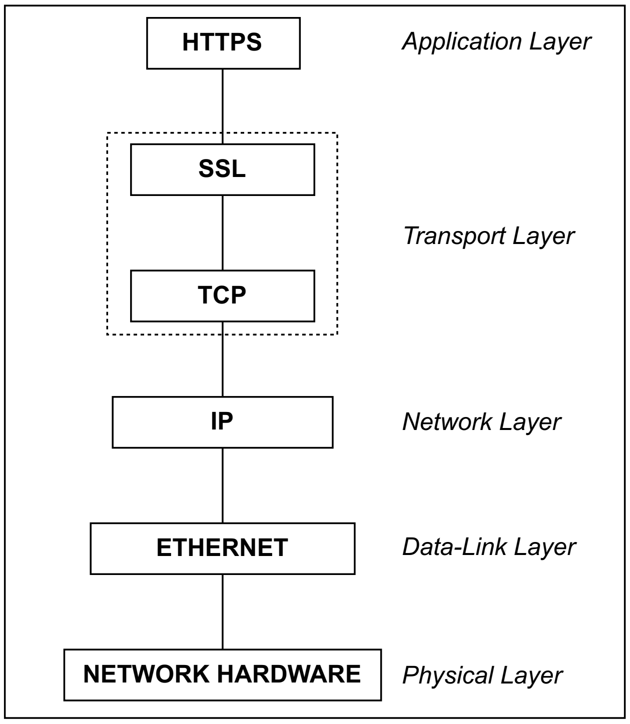
- The Internet-enabled vending machine can now become a reality—SSL makes tampering with communications almost impossible.
- Home automation systems can be Internet-enabled—forgot to turn off the oven? Just log into your house from your computer at work and turn it off. SSL provides a secure means of protecting your home from hackers.
- Readings from medical devices can be sent over a standard network—SSL protects your privacy.
- Make a telephone switch Web-configurable—SSL encrypts all data, so no one monitoring the network can read your information. Since Web-based access means that your data will likely be travelling over a competitor's network, SSL makes a lot of sense.
- Remote-entry configuration—change the passcode on all the doors of a building simultaneously. SSL protects the passcode, allowing the doors to be connected to a standard corporate network, no need for expensive proprietary hardware!

i. At the time of this writing, HTTP file upload over an SSL-secured channel is not supported.

- Television cable box monitoring/billing—connect a cable box to the Internet to monitor use and provide online billing.
- Utilities monitoring/billing (gas, electric, water)—connect gas and electric meters to the Internet without the worry of users tampering with the information sent.

SSL is designed to run over TCP/IP. Figure 1 shows how the SSL protocol fits into the overall TCP/IP reference model.

Figure 1. How SSL Fits Into the 5-Layer TCP/IP Reference Model



What can SSL do for my application?

SSL protects the communications channel. It also provides authentication (on the client side, optionally on the server side) of communicating parties. SSL can secure any connection between two points, and no one monitoring the connection can do anything destructive or gain unauthorized access to any sensitive information. SSL provides a secure channel without the need for either end to meet to exchange keys. SSL is to secure communications as TCP is to normal communications—it provides a standard communications infrastructure that compliant applications can use easily and nearly invisibly.

SSL provides a vitally important component of any secure system. Basic authentication mechanisms such as the Telnet password and basic HTTP authentication become very powerful security options when executed using SSL instead of plain TCP—passwords are no longer sent plain-text, making these methods much more useful. SSL encrypts the *connection*, not the data at either end, and does not contain any mechanism for user authentication or password protection (only the connection is authenticated—the security fails if the machine at either end is compromised).

1.1 Document Overview

This document is arranged to help you start developing applications with SSL right away. It also provides more detailed information for interested advanced users. First, Section 2.0 describes the hardware and software requirements for developing with SSL. Section 3.0 gives a brief history of SSL for context, and Section 4.0 is a complete start-to-finish walkthrough that details the creation of an SSL certificate and setting up a sample SSL application using HTTPS. A list of sample applications is given in Section 5.0.

More advanced users with an interest in SSL can continue with Section 6.0, which covers the basics of the SSL protocol, and Section 7.0, which covers some specific Rabbit SSL details. Section 8.0 is a detailed reference for the SSL certificate utility that covers the advanced interface in detail (the wizard interface is covered in the walkthrough).

The appendices are primarily for reference. Appendix A introduces cryptography and some of the principles behind the design of SSL. Appendix B gives an in-depth explanation of SSL digital certificates, which will help advanced users with certificate management and use. Appendix C is an API reference for the new HTTPS socket wrapper functions, which are provided for direct socket access (for CGI) in SSL/HTTPS applications.

2.0 Hardware and Software Requirements

2.1 Rabbit SSL Requirements and Performance Characteristics

The following minimum requirements must be met to run SSL on a Rabbit microprocessor.

- Rev. A or later of the Rabbit 3000 microprocessor
- Dynamic C v. 8.30 or better (to support Rev. A of the Rabbit 3000)
- Minimum 256KB flash (512KB recommended)
- Minimum 256KB RAM (512 KB recommended)
- Network connectivity with TCP/IP
- Single thread only with 2–4KB stack in µC/OS applications (some SSL functions are not re-entrant)

The Rabbit SSL implementation is about 45KB of code, not including the TCP/IP stack (the size varies, depending on the options enabled). It uses 2–3KB of root memory and about 24KB of xmem for each connection (one connection is usually enough). SSL uses at most 1KB of stack space, so any application running SSL should use a minimum 2KB stack.

The SSL performance is quite high with a Rabbit microprocessor, reaching a throughput of 120 kbits/s on average on a Rabbit 3000 (Rev. A) running at 44 MHz (be aware that SSL imposes a large overhead to all communications). Session establishment—the longest operation of any SSL connection—with the Rabbit 3000 hardware assist takes about 3 seconds, which is quite fast, considering many PC-based implementations can take almost as long for the same operation.

2.2 Implementation with Dynamic C

One of the primary uses of SSL on the Rabbit is to secure HTTP communications by providing an HTTPS server implementation to communicate with standard Web browsers. Dynamic C's implementation of SSL should work with any browser that supports SSLv3. Browsers earlier than Internet Explorer version 4 and Netscape version 5 are not officially supported, but may work with the Rabbit SSL implementation. TLS is only supported on newer browsers—starting with Internet Explorer version 5 and Netscape Communicator version 6. Officially, Dynamic C's implementation of SSL is supported only on browsers newer than Internet Explorer version 5 and Netscape Communicator version 6.

3.0 History of SSL and TLS

The original Secure Sockets Layer (SSL) implementation was developed in the early 1990s by the Netscape Communications Corporation to secure HTTP, which sends its data as plain text over the Internet. The first official release was version 2.0, which gained widespread acceptance despite some design problems with the protocol.

In the late 1990s it became apparent that SSL 2.0 was not secure. Netscape began working on SSL 3.0. In conjunction with Netscape, the Internet Engineering Task Force (IETF, the Internet standards governing body) began work on standardizing SSL, a project that became known as TLS (Transport Layer Security). SSL 3.0 was not developed as rigorously as TLS, so it became available sooner and quickly overtook SSL 2.0 as the industry standard.

TLS was finalized in 2000, providing the first standardized protocol for SSL. Although SSL 3.0 is still in widespread use, it is mostly obsolete for new development since almost all modern browsers support TLS.

4.0 SSL Walkthrough

This walkthrough explains the setup and execution of a simple HTTPS server on a Rabbit-based device. There are six steps, which are listed here.

1. Create a digital certificate.
2. Import the certificate.
3. Set up TCP/IP for the sample application.
4. Set up the application to use SSL.
5. Set up the Web browser.
6. Run the application.

4.1 Create a Digital Certificate

The first step in setting up an SSL-enabled server is to create your SSL server certificate. This can be done using the Rabbit SSL certificate utility, which is included in your SSL installation. The utility can be found in the `Utilities\SSL_Utilities` directory under the root Dynamic C installation directory. Start the SSL certificate utility by double-clicking `certificate.exe`**.** This walkthrough shows you how to create your own Certificate Authority (CA) and its accompanying root CA certificate,[i] and how to create a server certificate signed by that root CA. This is only one of the three types of certificates that can be created using the certificate utility; see Appendix B. "SSL Certificates," for more information on certificates.

This walkthrough uses the certificate utility wizard interface. See Section 8.3 for further information about the advanced interface.

1. Start the wizard and set the output directory.
The utility should start in the wizard mode. This is the primary certificate utility interface, and it contains everything you need to create your own certificates. The advanced interface is accessed by clicking "Launch Advanced Interface" on the first wizard panel. For this walkthrough, just click "Next."

The second panel displays the output directory edit box. It will default to the directory you executed the utility from, and you can change the directory by clicking "Browse." Set your output directory to be the `Samples\TCPIP\SSL\HTTPS\cert` directory (relative to your Dynamic C directory) for the sample programs, and click "Next." Note that several files will be generated for each certificate, so it is a good idea to use an empty directory for your certificate output.

2. Select the signing options and create a root CA certificate.
Next, you need to select a signing option. Click the "Create your own CA" radio button on the signing options panel, then click "Next." You should see the "Create a New Certificate Authority" panel, where you can enter the information for your new Certificate Authority (CA) that will be stored in your root CA certificate. Enter "ca" in the Certificate Name field. Note that the certificate name is going to be used as a base name for the certificate files, so this should only contain characters that are valid in file names. For the Common Name, enter "My Root CA Certificate". You can fill in the optional fields if desired, then click "Next."

Now you must select an output directory for your new root CA certificate—select the "private" directory (a subdirectory under the directory where the utility executable is located), and click "Next." The root CA certificate will be created when you finish the wizard. This is the certificate that you will need to install in your browser later in order for the browser to accept the server certificate you will create next. Remember where your root CA certificate is for the browser setup.

3. Create the device server certificate.
Now you will create the certificate that will be loaded onto your device. For the Certificate Name, enter "mycert". Again, this will be used to generate file names, so only enter valid file name characters. For the Common Name, enter the IP address of your device (e.g., 10.10.6.100), or if you access your device using a text URL, enter the base Domain Name you access your device from. For example, if you access your device from "http://www.foo.com/index.html", your certificate's Common Name should be "www.foo.com"—no slashes or extra information.

Again, fill in the optional fields as desired, and then check the checkbox at the bottom of the panel to generate the certificate and exit the utility when you are finished. Click "Next."

4. Create the certificates.
Now that you have finished the wizard, you need to generate your certificates. Check the summary on the final panel of the wizard to make sure all the information is correct. Once you have verified your certificate information, click "Finish" to generate your root CA and server certificates.

---

i. A Certificate Authority (CA) provides the service of verifying the identity of a certificate owner using conventional means, then signs that owner's certificate using their own private key. The root CA certificate is a self-signed certificate that originates from a trusted source, and represents the ultimate guarantee that a certificate being checked is genuine. See Appendix B. "SSL Certificates," for more information.

The generated certificate will be saved in the output directory you selected in Step 1, and will be called *mycert.dcc* in the output directory you selected (assuming the file name you chose for your certificate was *mycert*). There will also be several other files generated along with the Dynamic C Certificate file. Most notably, the file *mycert.der* contains the browser-compatible version of the server certificate (you will not need this file unless you want to install your server certificate into your browser, which is not necessary but may be desirable in some cases).[i] The files *mycert.key* and *mycertkey.pem* are two formats of the private key, and these should be kept secret to preserve the security of your certificate (it is a good idea to protect these with some form of encryption). The file *mycerts.pem* is the signed PEM format version of your certificate (the 's' is added so it will not be confused with the certificate request *mycert.pem*). The *mycert.dcc* file will be imported into your Dynamic C application.

4.2 Import the Certificate

Now that you have created your certificate, it's time to open the sample program to import your certificate into. Run Dynamic C, and open the sample program `SSL_STATIC.C` in the `Samples\TCPIP\SSL\` directory. Locate the `#ximport` for the certificate import, and modify the line to point to your certificate. As long as you follow the above instructions to generate your certificate, you should ***not*** have to modify the `#ximport` statement.

It is also possible to store the certificate in the flash file system. It is up to the user, however, to implement *how* the certificate gets into the file system. See Section 8.0 "SSL Certificate Utility," for more information on certificate import options.

4.3 Set Up TCP/IP for the Sample Application

SSL is a security complement for TCP/IP. As such, SSL requires a TCP/IP connection in order to function. For this reason, you must set up TCP/IP for your device. Open the `TCP_CONFIG.LIB` library and follow the instructions provided in the library to set up your device for use with TCP/IP. Check that the device's address matches the Common Name of the certificate you created above. If the device is accessed through a proxy or DNS, then the certificate does not need to match the IP address since the browser will use the external address to access the device—however, the certificate's Common Name must match that address.

The sample application (`SSL_STATIC.C`) uses SSL through HTTPS, so we need to make sure the HTTPS port is enabled. Where normal HTTP uses TCP port 80, the standard HTTPS port is TCP port 443. Looking at the sample program, you will see that there is a call to `tcp_reserveport(443)` just below the call to `tcp_reserveport(80)`. This assures that both the HTTP and HTTPS ports are reserved for the server.

i. You will not need your *mycert.der* certificate unless you want to install your server certificate in your browser. If you follow the walkthrough, then you will be installing your root CA certificate, which will provide the trust for your server certificate. However, if you create a self-signed server certificate, or if you want the browser to just trust that particular device, then you would want to install the server certificate to avoid warnings from the browser.

4.4 Set Up the Application to use SSL

1. HTTPS configuration.
The application needs to know that some servers will use HTTPS. The use of HTTPS is controlled by a few macros, some of which will be familiar to you if you have already worked with the `HTTP.LIB` library.

The most important macro is `USE_HTTP_SSL`, which enables the use of SSL for HTTP servers (thereby making them potential HTTPS servers). The number of HTTPS servers used is determined by a combination of two macros, `HTTP_MAXSERVERS` and `HTTP_SSL_SOCKETS`. The `HTTP_MAXSERVERS` macro specifies the *total* number of HTTP *and* HTTPS servers to be used by the application. The `HTTP_SSL_SOCKETS` macro specifies the number of servers *out of the total* to secure with SSL. In other words, the number of HTTPS servers is determined by `HTTP_SSL_SOCKETS`, which cannot be greater than `HTTP_MAXSERVERS`**.** The number of plain HTTP servers is then given by the difference between `HTTP_MAXSERVERS` and `HTTP_SSL_SOCKETS`.

For example, say we want two normal HTTP servers and one HTTPS server. This would be done by using the following code in your program.

```
#define HTTP_MAXSERVERS 3    // There are 3 total servers

#define USE_HTTP_SSL         // Tell HTTP.LIB to use SSL
// Tell HTTP.LIB to reserve 1 of the 3 servers for HTTPS
#define HTTP_SSL_SOCKETS 1
```

In the example application, there are two servers enabled, with one of them reserved for HTTPS. This means that incoming requests will be handled according to the port they come in on. Requests on port 80 will be handled by the normal HTTP server, whereas requests on port 443 will be handled by the HTTPS server and will be secured using SSL.

2. Resource protection.
Now the application will accept connections from anyone, secured or not. While this may be acceptable in some situations, there is no way to force the client to use the secured port to communicate with the device, they can log in using plain HTTP. For this reason, Dynamic C SSL provides a mechanism to protect resources using the `ZSERVER.LIB` library. Any resources allocated using the `sspec_xxxx` functions with a server mask of `SERVER_HTTPS` will be accessible only via HTTPS, and will not load if accessed in any other way. See the `SSL_FORM.C` sample program for more information on resource protection.

The following example illustrates resource protection using `sspec_addform`. The form *myform* (simply a collection of 5 variables) is initialized in the call, with the name “myform.html” and a size of 5. The important thing to note is the server mask parameter, `SERVER_HTTPS`. This indicates to `HTTP.LIB` that this form can *only* be accessed using an HTTPS server. In this way, “myform.html” will only be accessible from the Web browser over an SSL-secured connection. Attempting to access the form without specifying HTTPS will result in a “404 Not Found” HTTP error.

```
// Declare form variables and form pointer
auto FormVar myform[5];
auto int form;

// Initialize the form and add it to the RAM resource list
   form = sspec_addform("myform.html", myform, 5, SERVER_HTTPS);
```

3. CGI programming with HTTPS.
CGI programs will benefit from the use of SSL and HTTPS. Applications cannot access the socket in an HTTPS server directly, so we have included some wrapper functions that allow the writing and reading of data directly to and from the socket. The `http_sock_xxxx` functions provide this functionality to developers. These functions use the socket associated with a particular HTTP or HTTPS server. If the server is an HTTP server, the socket used will be plain TCP/IP. If it is an HTTPS server, the socket will be secured with SSL. See Appendix C. “API Functions,” for more information.

4.5 Set Up the Web Browser

The sample program is now ready to compile and run. However, there are some steps to take to set up the browser before attempting to communicate with the Rabbit-based device. Some of the following steps are optional in the sense that not following them will not necessarily prevent communication from happening between the browser and the device, but will cause the browser to generate warnings.

1. Enable TLS 1.0 (optional).
Most modern browsers support the TLS standard. The IETF TLS 1.0 is the most up-to-date standard for SSL, replacing Netscape’s SSL version 3. By default, however, many browsers have TLS disabled, and only have SSLv3 enabled for SSL communication (newer browsers are starting to have TLS enabled by default). Rabbit’s SSL comes with both TLS and SSLv3 support, but TLS is considered more secure.[i] You can enable TLS by locating the security options in your browser and enabling TLS 1.0 (this option can be found in Microsoft Internet Explorer under Tools | Internet Options under the “Advanced” tab—just click the checkbox next to “Use TLS 1.0” if it is not already checked). If you are using only TLS in your application, and TLS is not enabled in the browser, the browser and the device will not be able to communicate.

2. Install your root CA certificate (optional).
In the first part of this walkthrough, you created your own Certificate Authority (CA) root certificate and used it to sign your server certificate, which at this point resides on your device (assuming you compiled and downloaded the sample program) in the path you selected when you created the certificate. The server certificate will be accepted by the browser if it meets certain criteria. These include: a valid digital signature, a valid date, a matching Common Name and URL address, and the certificate in the correct format. The first criterion is covered by the signing certificate (your root CA certificate in this case). The latter three criteria have also been covered (the default valid date range is one year from the date of creation, you matched the Common Name to the address when you created the certificate, and the certificate utility took care of the formatting).

If you chose “ca” for the name of your certificate, then the certificate file will be called *ca.der*. In Windows, this file type is associated with Internet Explorer, and you can install the certificate simply by opening it with a double-click. Opening the file will display the certificate dialog box, which will display information about the certificate. In the lower right corner of the window there is a button labeled “Install Certificate.” Click this button, and the Certificate Import Wizard will open. Click “Next.” The second panel will prompt you for a certificate store. Select the default option, click “Next” again, and then click “Finish.” The wizard will then ask you if you want to install the certificate in the root certificate store. If all the information looks correct, click “OK” and your certificate will be installed.

i. This is not necessarily true, as there is no evidence to show that SSLv3 is inherently less secure than TLS. However, because of the rigorous design requirements of the IETF, TLS can be *proved* to be more secure than SSLv3, and therefore is (intuitively) considered to be more secure.

You can verify that your certificate has been installed by opening Internet Explorer (version 6), and navigating to Tools | Internet Options..., and clicking the “Content” tab. Look for the “Certificate” button, click it, then click the “Trusted Root Certification Authorities” tab. Search the list for your certificate, and double-click it to view the certificate details. You can also use the list to remove your certificate.

> **WARNING:** DO NOT remove any certificates you did not install yourself unless you know what you are doing, since doing so will prevent you from viewing Web sites protected by certificates signed by those authorities.

Netscape and other browsers use different methods for installing root certificates. Refer to the documentation for each of those browsers for instructions on installing root certificates. If you have problems installing your root CA certificate in a specific browser, please contact your distributor or Rabbit partner, or use Rabbit’s Tech Support e-mail form at www.rabbit.com/support/.

4.5.1 The https Prefix

The https prefix is not really a browser setup issue, but you should remember to use the https (note the ‘s’) prefix at the beginning of the URL to access HTTPS pages. For example, if your device is accessed from www.foo.com, the complete address would then be ***https***://www.foo.com. The https prefix tells the browser to make requests on the HTTPS TCP port 443 instead of the standard HTTP port 80.

4.6 Run the Application

Once the sample program has been compiled, downloaded, and started, you can access your device from your browser by typing “https://<my device URL>” where <my device URL> is the IP address or URL of your device.

4.6.1 Expected Behavior

If you are running the `SSL_STATIC.C` sample program, the SSL handshake process begins once you type in the address of the device and hit enter. If you followed the walkthrough and installed the proper root CA certificate in your browser (and your server certificate matches the device’s address), then you should see the “lock” icon appear in the lower right corner of your browser. The static page should then display. You can double-click the lock icon to look at your certificate, which will bring up the certificate browser. You can look to see that all the information is there and correct, and you can also follow the certification path (the “Certification Path” tab) back to your installed root CA certificate.

If you did not match the common name of your server certificate to the device address (Step 3 in Section 4.1) or if you did not install your root CA certificate (Step 2 in Section 4.5), the browser will issue a warning indicating a problem with the certificate, along with details about what has failed. It will allow you to either cancel the page load or ignore the warning. Since it is your device, you can just ignore the warning and continue to the page load. The page is still secured with SSL—you have simply opted to trust the device’s certificate (and therefore its identity) despite the browser’s warnings of a possible problem.

4.6.2 Troubleshooting

If you followed the walkthrough and your page is not displaying or there are other problems, the following tips may help you to troubleshoot the problem.

- *Check that TLS 1.0 is enabled in your browser.* This is one of the most likely causes of problems, since the browser will simply not accept the page if the protocol is wrong. If you are using TLS only in your application and your page is not loading, make sure TLS is enabled in your browser.
- Make sure you are using *https://* and not *http://*. One of the most common causes of problems with loading HTTPS pages from a Rabbit-based device is one of the most obvious. Make sure that the URL in your browser starts with https:// and not http://.
- *Check that your TCP/IP settings are correct.* This may seem obvious, but since SSL relies on TCP/IP for the network connection, if your TCP/IP settings are not correct, SSL will not work.
- *Make sure your proxy is configured correctly.* If your device is behind a proxy, you may need to configure the proxy to forward port 443 for HTTPS.
- *Check the certificate.* If the certificate is not properly formed, or was signed by an unknown Certificate Authority, the browser may simply fail to load the page. First, check that your root CA certificate is correctly installed in your browser. Also, check that your root CA certificate does not have a Common Name (CN) that matches any other certificate in the root certificate store. If it does, any certificates you have signed using it will be rejected by the browser as having an invalid digital signature. You can also look at your server certificate directly. It will be found along with your DCC file, with the extension DER. Double-click the DER file to bring up the certificate browser to allow you to verify that the certificate is correct.

5.0 SSL Sample Programs

The `SAMPLES\TCPIP\SSL\HTTPS` folder provides sample programs that illustrate various aspects of SSL. Each sample program has comments that describe the purpose and function of the program. See the comments at the beginning of each sample program for more information.

To run a sample program, open it in Dynamic C, then press the F9 function key to compile and run it. The Rabbit-based device must be connected to a PC using the programming cable.

- `SSL_AUTHENTICATION.C`—This sample program demonstrates HTTP authentication over HTTPS, which encrypts both the authentication data (username and password) and the Web page itself.
- `SSL_CGI.C`–This sample demonstrates the use of CGI functionality over HTTPS with a page-hit counter.
- `SSL_FORM.C`—This sample program uses a sample thermostat control form to demonstrate secure form submission using HTTPS.
- `SSL_SSI.C`—This sample demonstrates server-side include (SSI) functionality by creating several "devices" (graphic icons) whose state can be changed simply by clicking on graphic buttons on the Web page.
- `SSL_STATIC.C`—This is the most basic SSL/HTTPS sample program. It simply establishes an SSL session and transmits a static HTML Web page to the browser over HTTPS.
- `SSL_ZIMPORT.C`—This sample program demonstrates HTTPS integrated with the `#zimport` functionality to compress and store a large text-based Web page and the ability to download that page securely over HTTPS.

6.0 SSL Basics

Although SSL is simple in theory (keys are exchanged using public-key cryptography, communication is done using symmetric-key cryptography), the actual implementation is quite complex. This section briefly covers the details of establishing an SSL connection and communicating using that connection.

6.1 SSL Alerts

One of the most important components of SSL is its error-handling system. SSL errors are called *alerts*, and represent possible attacks. Alerts are messages sent across the SSL communication channel, and may be encrypted. The SSL specification details about 20 different alerts and gives guidelines on how to handle them when received, and when to generate and send them. Error handling is implementation-specific, and will be covered later in this document.

6.2 The Handshake

SSL communication takes place in an SSL *session*. The session is established using a handshake process similar to the TCP 3-way handshake. The entire handshake, including the establishing the TCP/IP socket, is shown in Figure 2. As can be seen in the figure, the TCP/IP connection is established first, and then the SSL handshake begins. The SSL session is established when both the client and server are communicating using the negotiated parameters and ciphers. The SSL session ends when either side is done transmitting application data and notifies the other machine that it is done sending data. The following sections describe this process in more detail.

6.2.1 The Client Hello and Public-Key Operation

All SSL sessions begin with a *Client Hello* message. This message is sent by the client to the server it wishes to communicate with. This message contains the client's version of SSL, a random number used later in key derivation, as well as a collection of ciphersuite *offers*. The offers are identifiers that specify the ciphers and hashing algorithms the client is willing to use.

A possible offer could be `TLS_RSA_WITH_RC4_128_MD5`, which indicates that the client is willing to speak to the server using TLS, RSA for the public-key operation, RC4 with 128-bit keys for the symmetric cryptography, and `HMAC_MD5` to generate the message MAC (it uses HMAC since the offer is TLS).[i]

When establishing the initial connection, the server chooses an offer it is willing to use, and communicates that offer back to the client along with its certificate and a random value of its own. The client then verifies the server using the certificate and extracts the server's public key. Using the public key, the client encrypts the *pre-master secret*, a random value that will be used to generate the symmetric keys independently, and sends the encrypted message to the server, which decrypts the message using its private key.

i. Although the offer indicates TLS, the offer is compatible with SSLv3, and corresponds to SSLv3 using RSA, RC4-128, and MD5 (not HMAC).

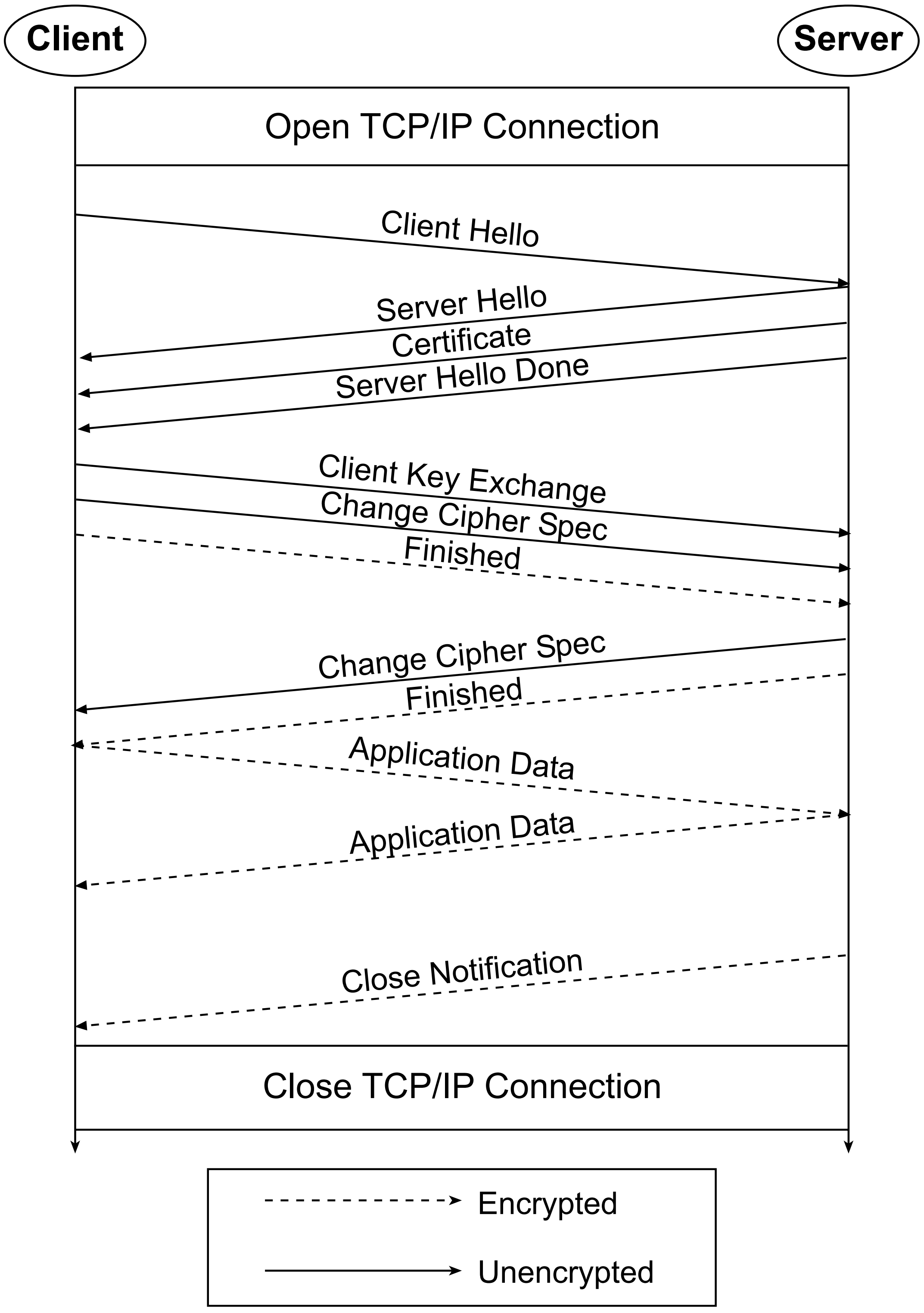


Figure 2. SSL Handshake

6.2.2 Symmetric-Key Derivation

Once the server receives the pre-master secret from the client, both the server and the client generate the *same* symmetric keys using the pre-master secret and the random numbers exchanged above using the TLS pseudo-random function (PRF), which expands a secret and some data into a block of arbitrary length.[i] This way, only the small pre-master secret is encrypted using public-key cryptography, limiting the impact of the expensive operation on performance.

6.2.3 Handshake Finish

As soon as the keys are generated, the client and server exchange *change cipher spec* messages to indicate that they each now have symmetric keys and all further communications will be conducted using the symmetric algorithm chosen in the initial stages of the handshake. At this point, the server and client take all the handshake messages received and sent, and generate a block of data used to verify that the handshake was not tampered with. These data, generated using the TLS PRF, are sent in a final handshake message, *Finish*. If the data in the finish message do not match the locally generated finish data, then the connection is terminated by whoever failed the finish verification test.

6.3 SSL Session

Once the handshake is finished, the server and client begin to communicate over the newly established secure channel. Each message is hashed, encrypted, and sent. If at anytime there is a failure, either in the decryption, encryption, hashing, verification, or communication, an SSL alert is sent (using the symmetric encryption) by the entity experiencing the failure. Most alerts are fatal, causing the communication to stop immediately.

6.4 Ending the Session

When the client or server is done communicating, a special alert, *close_notify*, is sent to ensure that all communications have ceased and the connection can be closed. This alert prevents an adversary from performing a *truncation attack*, fooling the server or client into thinking that all the data to be exchanged have been sent, when actually there are some data left (this can be a problem in situations such as banking transactions, where it is necessary for *all* information to be received).

6.5 SSL Security

The elaborate and expensive establishment of an SSL session is the result of many years of studying the various attacks performed against SSL and other secure protocols. Some attacks target the cryptography implementations, others target the PRNG. Some attacks can use knowledge of the information being sent to derive secret information. Some attacks even use the timing of certain algorithms to derive secrets.

SSL and TLS address many of these issues in its protocol design, but even more importantly, implementations must follow the protocol for those safeguards to work. Beyond that, implementations must also not be subject to other forms of attack, such as buffer overflow.

---

i. SSLv3 uses its own algorithm to generate the key material. The algorithm is based on an early version of HMAC, and has no known security vulnerabilities.

7.0 SSL On the Rabbit

7.1 SSL Performance

SSL performance is dictated by cryptography and hashing. Rev. A (and newer) Rabbit 3000 chips (and newer microprocessors) contain special instructions that speed up the RSA operation significantly, with the operation taking just over 2 seconds for a 512-bit key (the smallest size any browser will accept) on a 44 MHz RCM3200. With the RSA operation, the entire handshake takes approximately 2.5 to 3 seconds.

The SSL implementation currently uses the RC4 stream cipher, which is simple and quite fast for a cipher, although at 15 KB/s (including the SSL overhead) it is considerably slower than a raw connection.The RC4 algorithm is the fastest and smallest of the cipher algorithms, and therefore is ideal for embedded processors.

7.2 Memory Usage

With SSL, every message received must be parsed and processed, which uses a lot of buffer space. Every message must be hashed, and the keys must be generated (one key each for read and write, client and server, a total of 4 keys for stream ciphers—block ciphers require additional keys). The certificates must also be stored, which can be as much as 1KB per certificate (some applications can require multiple certificates).

The current SSL implementation stores the certificate as a large constant in flash or XMEM, or as a file in the flash file system.

7.3 Error Handling

One important aspect of the SSL implementation is its ability to handle errors. Errors are especially important because they represent possible attacks, and therefore need to be handled appropriately to prevent the compromise or loss of data. In addition to the SSL alert mechanism (used to communicate errors between the client and server), the Rabbit SSL implementation uses a global error number, which is similar to the FS2 `errno`. The `SSL_errno` is set to one of about 30 predefined values that identify the most recent error. A user can use this information to take appropriate action when an error occurs.

7.4 SSL-Specific Considerations

The SSL standards define what ciphersuites and features must be included for full compliance.

First, SSL *must* be implemented over a communications channel such as TCP that will assure all data are received in order they were sent. An unreliable protocol such as UDP *will not work* with SSL.

The TLS RFC states that certain algorithms are mandatory for an implementation to be fully TLS-compliant. Diffe-Hellman, DSS (Digital Signature Standard), and the 3DES cipher are examples of such algorithms. However, these algorithms are not necessary for the TLS communication itself since nearly all TLS implementations already support RSA, MD5/SHA-1, and bulk ciphers other than 3DES, such as RC4. For performance reasons, the Rabbit SSL currently supports RC4 for bulk encryption and RSA for public-key and digital signing operations. These algorithms have definite performance advantages over Diffie-Hellman and 3DES, and are more than adequate to be compatible with the vast majority of SSL client implementations.

The TLS RFC and SSLv3 specification also describe some “advanced” features that may be necessary for communication. The first of these is the SSLv2 backward-compatible handshake. SSLv2 uses a completely different message format to establish a connection. For backward compatibility, most SSLv3 and TLS implementations can recognize an SSLv2 client hello message. The contents of the SSLv2 message can be mapped directly to an SSLv3/TLS client hello, and parsed by the server. The server will reply to the client hello with an SSLv3 or TLS handshake message, effectively upgrading the security from SSLv2 to a newer protocol, or causing an error if the server is actually an SSLv2 application. Nearly all browsers will use the SSLv2 client hello to initiate communication. Rabbit SSL supports SSLv2 backward compatibility (note that this does not mean full SSLv2 support).

The SSL standards also describe how to perform *session renegotiation*, a very important SSL feature. Session renegotiation allows a client and server who have previously negotiated an SSL connection to continue where they left off after a break in communication (which may include closing the TCP/IP socket connection). SSL achieves this by using large numbers called *session identifiers* (session IDs) negotiated in the initial handshake. The server and client save all the required information (such as cipher state and the master secret for key derivation) associated with a session when a connection goes down (that is, the socket is closed or reset). This information is stored with the session ID. When the client re-initiates contact, the session ID identifies the client, and the existing state is used, thereby avoiding the costly public-key operation. This can improve performance dramatically for applications such as Web servers, where connections are opened and closed frequently. Security is not affected by session renegotiation unless either entity is compromised, since both the server and the client already have the secret information known only by them (the symmetric keys have already been exchanged). Rabbit SSL fully supports session renegotiation.

8.0 SSL Certificate Utility

8.1 Overview

Each device running SSL needs a certificate to be able to communicate with Web browsers and other SSL implementations. The certificate utility helps you create SSL certificates for your devices. The utility can be found in the `Utilities\SSL_Utilities` directory in the root Dynamic C installation. Start the certificate utility by double-clicking `certificate.exe`.

The SSL walk-through in Section 4.0 detailed the creation of a Certificate Authority (CA) and its root CA certificate, and the creation of a server certificate signed by that root CA using the wizard interface. This section covers all aspects of the certificate utility, and is intended as a reference. For more information on certificates and choosing signing options, see Appendix B. “SSL Certificates.”

8.2 The Wizard Interface

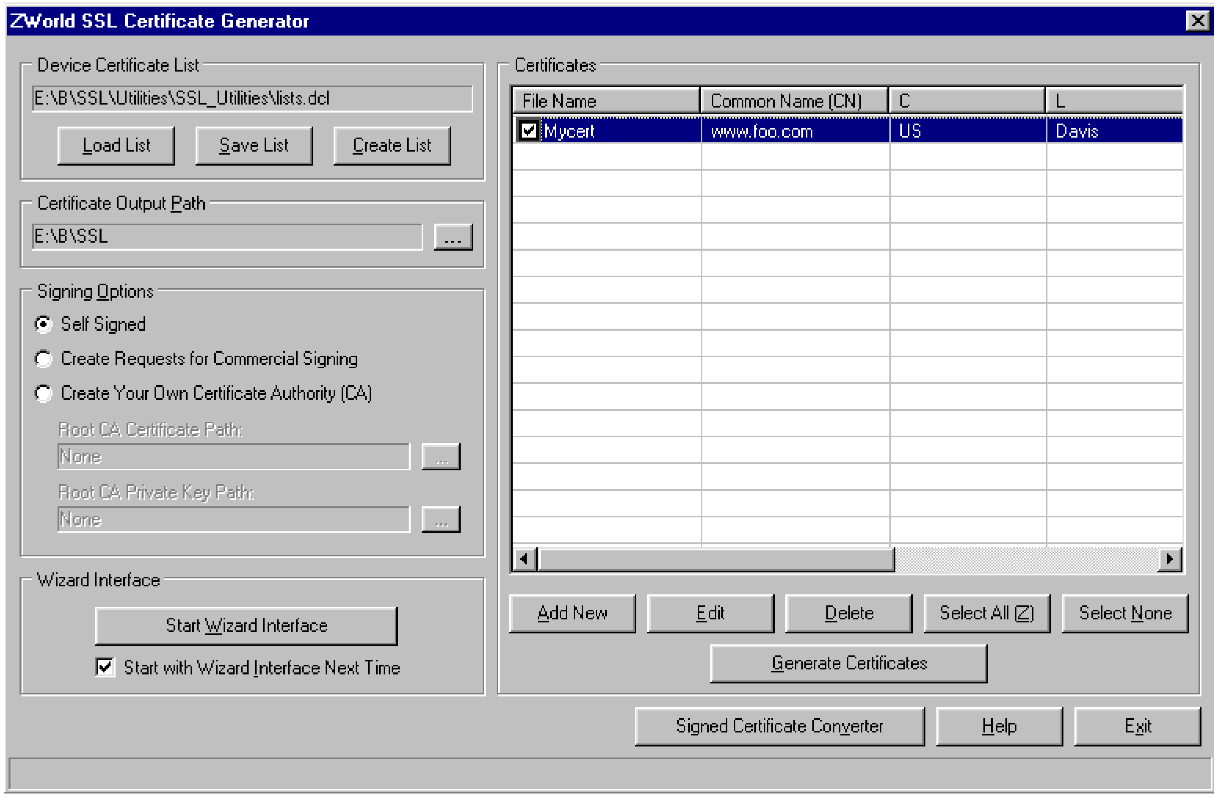
When you first execute the certificate utility, you will be presented with the wizard interface. This interface takes you step-by-step through the certificate creation process. Follow the on-screen instructions to create your certificates, or set up the advanced interface with a root CA certificate and certificate list (more information on the advanced interface follows). If you choose, you can also skip directly to the advanced interface from the first wizard panel by clicking on the “Launch Advanced Interface” button. Note that if you want to create your own root CA certificate, you will have to go through the entire wizard (be sure to select “Create a New Certificate Authority” on the signing options panel).

You also have the option to create a certificate and exit the wizard, or create a certificate to be added to a certificate list for the advanced interface. Checking the “Generate this certificate at finish and exit” checkbox on the “Create a New Certificate” panel will exit the utility once your certificate is generated when you click “Finish.” If you leave the box unchecked, the advanced interface will start with all the options you selected with the wizard present after you click “Finish.”

8.3 The Advanced Interface

The advanced interface is illustrated in Figure 3. The file and signing options are on the left-hand side of the window, and the certificate edit pane is on the right.

Figure 3. The Primary Certificate Utility Window



Device Certificate List: E:\B\SSL\Utilities\SSL_Utilities\lists.dcl

Certificate Output Path: E:\B\SSL

Signing Options: [✓ Self Signed] / Create Requests for Commercial Signing / Create Your Own Certificate Authority (CA)

Root CA Certificate Path: None

Root CA Private Key Path: None

Wizard Interface: [✓ Start with Wizard Interface Next Time]

Certificates

File Name	Common Name (CN)	C	L
✓ Mycert	www.foo.com	US	Davis

8.4 Creating and Loading Certificate Lists

To begin the certificate creation process, you need to create or load a new certificate list. If you set up the advanced interface using the wizard (you did not check the “Generate Certificate and Exit” checkbox), you can skip the following steps since the wizard already filled in all the file and signing options for you. Also, the utility stores all your choices on exiting, so the next time you start the utility, you are ready to go.

The certificate list is simply a file that contains the certificate information for one or more certificates. You can create a certificate list by clicking the “Create List” button in the top left area of the window. A dialog box will open to allow you to select a file name for the list. You can also choose to load a previously saved list using the “Load List” button. After you chose a file, the entire path of the current list is displayed in the Device Certificate List text box (if your path is too long to fit into the text box, placing the mouse cursor over the box for a few seconds will display the entire path for you). Note that certificate lists are stored in the Dynamic C Certificate List (DCL) file format.

Once you are done editing your certificate list, click the “Save List” button to preserve your changes.

You will also need to set up your output directory. You can do this by clicking the “...” button next to the text box under “Certificate Output Path” on the left side of the window. Note that several files will be generated for each certificate, so it is a good idea to use an empty directory for your certificate output.

8.4.1 Signing Options

You need to select a signing option for your certificate. This is done using the radio buttons on the left side of the window.

Note that the default signing option is “Self Signed.” This will generate certificates that are signed using their own private keys. These certificates can be used for testing, or if the user does not mind the warning messages generated by browsers when attempting to connect to a device. Note that these certificates may be installed in a Web browser’s certificate store to eliminate the warnings (see Section 4.5 “Set Up the Web Browser,” for more information). If you are deploying a large number of devices each with different certificates, then the “Create Your Own CA” option may be a better choice.

The second option is “Create Requests for Commercial Signing.” This option allows you to create certificate requests for sending to a commercial Certificate Authority (CA) for signing. You can than convert the signed certificate you receive from the CA to a Dynamic C Certificate (DCC) file using the “Signed Certificate Converter” button (see below for more information). This option has the advantage of your certificate being automatically trusted by any Web browser or implementation (assuming you used a trusted CA to sign your certificate for you). Use this option if your device will be accessible on the Internet and you want any browser to trust the certificate immediately.

The third and final signing option is “Create Your Own CA.” Upon clicking the radio button, the root CA certificate and key paths will become enabled, allowing you to load a root CA certificate and its matching key.

To create a new root CA certificate, use the wizard interface and select the “Create a New Certificate Authority” option on the signing options panel, and leave the “Generate this Certificate at Finish and Exit” checkbox unchecked. Once you finish the wizard, you will see that the root CA certificate and key paths now point to your newly created root certificate and key. The DER version of this certificate is what you will need to install in your browser later for the browser to accept the server certificate. Remember where your root CA certificate is for the browser setup.

If you have already created a root CA certificate that you want to use, then you will need to load that certificate and that root certificate’s matching private key file (both in PEM format) using the “...” buttons next to the respective path text boxes for the certificate and key.

8.4.2 The Certificate Edit Pane

The certificate edit pane (labeled Certificates, see Figure 3) shows certificates in the current list that may be edited or generated. The fields for each certificate are represented by columns, with each row representing a single certificate. Before you choose a list, this pane and the associated buttons will be disabled. Creating or loading a new list will enable the edit pane, and you will be able to add, edit, and delete certificates from the list. In Figure 3, a simple list has been loaded, showing the file name and Common Name of a single certificate.

To edit a certificate, either double-click it or highlight the certificate and press enter or click “Edit.”

To create a new certificate, click the “Add New” button. This will bring up the certificate edit dialog window. Fill in the mandatory and desired optional fields (described in Section 8.4.4, “The Certificate Edit Dialog”) and click “OK” to add the new certificate to the list. Note that the list is sorted alphabetically according to the file name, and may update after editing or creating a certificate.

You can remove certificates from the list by highlighting them and clicking “Delete,” or hitting the delete key (this cannot be undone).

8.4.3 Generating Certificates

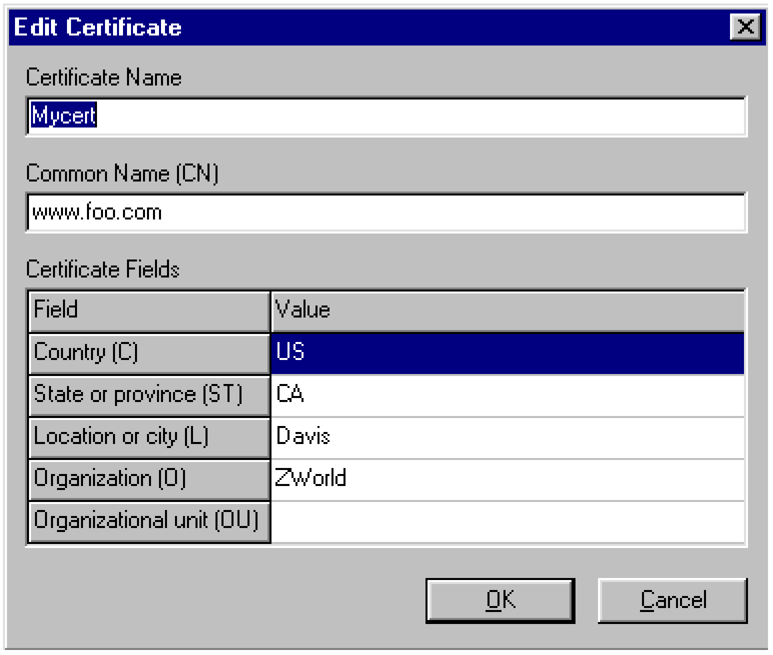
Once you are satisfied with your certificate list, you can generate your certificates. To do this, check the checkbox next to each certificate you wish to generate (the “Select All” and “Select None” buttons help to speed up this process), and click the “Generate Certificates” button. If all went well, a message box will be displayed, indicating success and the number of certificates generated.

The generated certificates will be saved in your chosen output directory, and will be named using the certificate name field of each certificate. Note that the name is the base name used for all the files generated for each certificate. The other files will be placed in the output directory along with the certificate. Most notably, the file *<certname>.der* contains the browser-compatible version of the server certificate (you will not need this file unless you want to install your server certificate into your browser, which is not necessary but may be desirable in some cases). The files *<certname>.key* and *<filename>key.pem* are two formats of the private key, and these should be kept secret to preserve the security of your certificate (it is a good idea to protect these with some form of encryption). The file *<certname>s.pem* is the signed PEM format version of your certificate (the ‘s’ is added so it will not be confused with the certificate request *<certname>.pem)*. The file *<certname>.dcc* file will be imported into your Dynamic C application.

8.4.4 The Certificate Edit Dialog

When you create a new server certificate or edit an existing certificate, you will use the Certificate Edit dialog.

Figure 4. Certificate Edit Dialog



Certificate Name: Mycert

Common Name (CN): www.foo.com

Certificate Fields

Field	Value
Country (C)	US
State or province (ST)	CA
Location or city (L)	Davis
Organization (O)	ZWorld
Organizational unit (OU)	

The only mandatory fields are the file name and the Common Name. The file name represents the base file name that will be used for all the generated files related to that certificate, so it should not include an extension or any characters (such as punctuation) that cannot be part of file names.

Choose the Common Name to match the domain you will access the device from, such as www.foo.com. Do not include pages or other information. The Common Name may also be the IP address of the device, if that is how the device will be accessed. The important thing is that the Common Name matches the address you will access the device from in your browser.

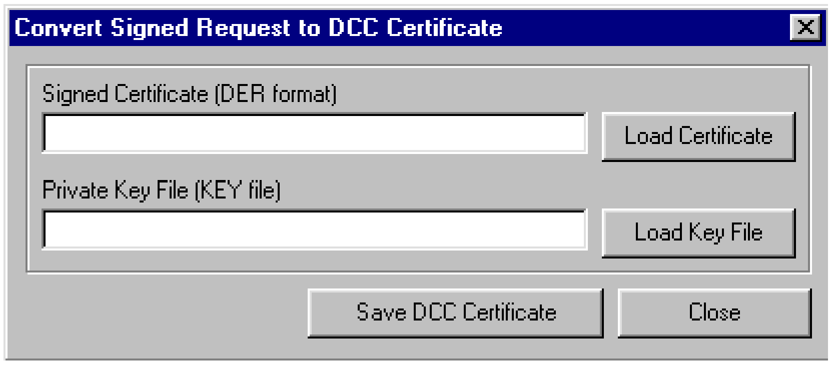
The other fields are optional—you can fill them in as desired. Note that these are the only certificate fields supported by the utility at this time.

8.4.5 The Signed Certificate Converter

If you chose the "Create Requests for Commercial Signing" signing option, you create certificate requests that are then sent to the commercial CA for signing. You should receive a signed certificate from your CA in the form of one or more files. One of these files should be the DER-formatted version of your signed certificate. In order for Dynamic C to be able to use your certificate, you need to create a Dynamic C Certificate (DCC) file by combining your DER certificate and the KEY file generated when you created your original request.

To create a DCC certificate file, click the "Signed Certificate Converter" button in the utility to bring up the Signed Certificate Converter dialog.

Figure 5. Signed Certificate Converter Dialog


Convert Signed Request to DCC Certificate
Signed Certificate (DER format)
Load Certificate
Private Key File (KEY file)
Load Key File
Save DCC Certificate
Close

To create your DCC file, load your DER certificate file by clicking "Load Certificate" and its matching KEY file by clicking "Load Key File." Click "Save DCC Certificate" to select the file name and output directory for your DCC file.

8.5 OpenSSL Libraries

The certificate utility uses the OpenSSL libraries and a command-line utility for the back-end work of actually generating the certificates. Primarily, the utility uses two Rabbit utilities, `mk_ca.exe` and `mkreq.exe`, which are located in the same directory as `certificate.exe`. These do the actual generation of certificates using Dynamic C Certificate List (DCL) files. Note that these utilities are the only Rabbit code linked against the OpenSSL libraries. The GUI utility simply executes them as from a command line.

The utility installation directory also includes the OpenSSL DLLs and the OpenSSL command-line utility (used for signing certificates and converting between different certificate formats). See www.openssl.org/ for more information about OpenSSL.

The OpenSSL license agreement stipulates that the following disclaimers be present in our documentation since we are using OpenSSL to generate certificates.

> "This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit. (www.openssl.org/)."
>
> "This product includes cryptographic software written by Eric Young (eay@cryptsoft.com)."

> **NOTE:** No OpenSSL code is present in any of the Dynamic C SSL libraries or the certificate utility (which simply executes command-line executables that *are* linked with OpenSSL). The Dynamic C SSL libraries and the certificate GUI utility are 100% proprietary to Rabbit Semiconductor.

8.6 Cygwin

The command-line utilities were built using GCC under the Cygwin environment, and therefore require the presence of the Cygwin runtime library `Cygwin1.dll`.

Note that this may cause conflicts if you have Cygwin installed on your machine.

See http://www.cygwin.com for more information on Cygwin.

9.0 References

1. Bruce Schneier, *Applied Cryptography,* 2nd edition. The standard layman's text on cryptographic algorithms and techniques.
2. Eric Rescorla, *SSL and TLS, Designing and Building Secure Systems*. A book that specifically covers SSL and how to write an implementation. Includes a discussion of HTTPS.
3. IETF RFC2246. Draft specification for TLS v1.1.
4. IETF RFC2818. RFC specification for HTTPS.
5. IETF RFC2104. RFC specification for HMAC.
6. IETF RFC2202. RFC specification for HMAC test cases.
7. RSA Security Public-Key Cryptography Standards #1. Standards for using RSA encryption.
8. OpenSSL documentation. Open-source implementation of SSL/TLS.

Appendix A. Cryptography and Message Verification

This appendix introduces cryptography and some SSL fundamentals. It is intended as a reference for an interested user to gain some insight into the inner workings of SSL, which can also aid in application development and debugging. The first section covers cryptography basics and message digest algorithms. The following section is an overview of SSL principles and standards, followed by brief discussions about SSL and TCP/IP, and SSL and HTTP.

A.1 Cryptography

Cryptography is the science of encoding data such that the data cannot be easily recovered without knowing some secret key. Cryptography is as old as writing—there is evidence that ancient Romans and Egyptians had notions of cryptography and used them to protect military and political correspondence. Cryptography also forms the basis for all computer communications security today.

In order to provide a secure communication channel, SSL relies on a number of *ciphersuites*, collections of ciphers and hashing mechanisms to encrypt and verify data (we cover hashing below). The actual algorithms used by SSL are covered in the sections on *SSL Standards and SSL Basics*.

A.1.1 *Symmetric-Key* Cryptography

Symmetric-key cryptography is the oldest and most widely used way to encrypt data to hide it from an adversary. To use symmetric-key cryptography, both of the communicating entities need to know a single, secret, shared key, which is used to both encrypt and decrypt the data. Symmetric-key algorithms for computers are quite fast and usually quite simple. Common algorithms such as RC4 (developed by RSA Security) can be implemented in just a few lines of C code. Other common symmetric-key algorithms include DES, 3DES, and AES.

A.1.2 *Public-Key* Cryptography

The problem with symmetric-key cryptography is that there needs to be a way to share the key between the communicating entities before they can communicate securely. Historically, this has been done using trusted messengers who hand-carry the key from one side to the other. This, however, is not an acceptable solution for computer communications, and so SSL employs a *public-key* algorithm.

Developed in the 1970s, public-key algorithms use two separate keys, one to encrypt and the other to decrypt. For this reason, public-key cryptography is also called *asymmetric cryptography* (hence symmetric-key cryptography). This way, the public key can be sent in plain-text over the network, and the client can encrypt data, knowing that only the person with the paired key can decrypt it.

It would be possible to implement a secure channel using only public-key cryptography. Each entity would have a key pair, and the public keys would be exchanged via plain text and used to encrypt messages back and forth. However, the problem is that public-key algorithms are excessively slow since they are based on multiplication that involves 100-digit-plus numbers, which requires millions of operations. They are so slow, in fact, that many companies implement public-key algorithms in hardware, where they can still take up to half a second!

SSL avoids the performance hit by using public-key operations only during the initial handshake to exchange a single message containing a secret. This secret is then used to generate symmetric keys that are used with the speedier symmetric-key algorithms. Common public-key algorithms include RSA and Diffe-Hellman.

A.2 The Importance of Randomness

SSL security relies on the fact that an adversary cannot decrypt messages without the key except by trying every possible key used to encrypt the data (for a 128-bit symmetric key, or a 1024-bit public key, this process would take hundreds or thousands of years using current technology). The problem is that we need to take precautions to ensure that the adversary cannot improve his chances at guessing the key. For this reason, all SSL symmetric keys are generated with the help of a *cryptographically secure pseudo-random number generator (PRNG)*. The PRNG normally uses an external entropy source such as circuit noise that is nearly impossible to predict. An early Netscape implementation of SSL 2.0 was compromised in just a few hours because the computer's date and time, clearly *not* truly unpredictable entropy sources, were used to seed the PRNG.

A.3 Message Verification with Hashing

In addition to hiding data using cryptography, SSL needs to verify that messages are not tampered with during transport. SSL achieves this through the use of *hashing* algorithms. For our purposes, a hash is a unique number generated directly from the data in a message. The idea behind message hashing is that it is impossible to generate the message from the hash, and hashing a message twice always produces the same value. For SSL, the important algorithms are SHA-1 (developed by the NSA) and MD5 (developed by Ron Rivest, the 'R' in RSA).

A.4 MD5, SHA-1, and HMAC

SSL uses the MD5 and SHA-1 hashing algorithms extensively. Most importantly, they are used to generate the Message Authentication Code (MAC) for each message. They are also used to verify the entire handshake and to generate the symmetric key.

For top performance, it is necessary that the implementations of MD5 and SHA-1 be as fast as possible.

TLS does not use MD5 and SHA-1 directly, but instead uses a keyed hashing scheme called HMAC. HMAC, which is presented in IETF RFC 2104, wraps a hashing algorithm and adds provable security properties to a hashing algorithm. SSLv.3 uses a variant of HMAC, and TLS uses HMAC itself for all message hashing and to generate the MAC.

A.5 P-HASH and PRF Algorithms

TLS requires two additional algorithms, which use HMAC, to generate the symmetric key. The first of these algorithms is P-HASH, which will expand a secret and a seed value into an arbitrarily large chunk of pseudo-random data using both HMAC-SHA1 and HMAC-MD5. Note that these data are pseudorandom and deterministic. The same secret and seed values will *always* generate exactly the same data.

The PRF algorithm is used to derive the symmetric key and in the handshake verification. It uses P-HASH to expand a secret into a chunk of data that can be divided into pieces for use as symmetric keys or as verification data.

A.6 An Overview of SSL

SSL is a secure protocol designed to run over TCP/IP and to be easily interchangeable with the standard UNIX-style sockets API used by almost all networking software. Security is provided by a combination of *Public Key* and *Symmetric Key* cryptography along with a *certificate* infrastructure.

A certificate is a collection of identification data in a standardized format. The data are used to verify the identity of an entity (such as a Web server) on the Internet. The certificate is digitally *signed* by a Certificate Authority (CA), which is a trusted entity given the power to verify an individual or company wishing to provide a SSL-secured application. Any client wishing to communicate securely with that entity can verify its identity by polling a CA certificate database (this process is described more in Section 5).

The certificate also contains the *public key* of its owner. This key matches a secret *private key* that only the owner knows. The key pair is used both to verify the identity of the certificate owner, and to allow secret information to be exchanged between the certificate owner and another entity. (This process is described below).

An SSL session consists of several parts, which are described in detail in Section 6.0, “SSL Basics.”

1. The *handshake*.

   The handshake is used to establish a secure channel between two entities. The client requests a secure connection from an SSL-enabled server by providing a message containing its parameters and requirements for the connection. The server replies with its certificate and its choice of SSL parameters. The client then verifies the certificate and the server’s identity before continuing. If the verification is successful, the client uses the public key extracted from the server’s certificate and encrypts a secret message and sends it to the server. The server then decrypts the secret, and both it and the client derive the *session keys* using the shared secret. The session keys are used with symmetric-key encryption algorithms (which are much faster than public-key algorithms) to send application data through the secure channel.

2. The *session*.

   Once the SSL handshake is completed, the session begins. The two entities now communicate data using the secure channel. When one side has finished, it sends a final message indicating that it is done sending messages and will close the connection. At this point the server caches the session information using a unique identifier established during the handshake. This information is used if the client attempts another communication to do what is called a *session renegotiation*.

   An important feature of SSL is its ability to do these session renegotiations. The session information cached by the server can be used to resume an SSL session where it left off, avoiding the expensive public-key operation. This is especially valuable for applications such as Web servers that may connect and reconnect many times (such as each time a user clicks a link on a Web page and is sent to a new page).

3. The *close*.

   When the client or server is done communicating, a special alert, *close_notify*, is sent to ensure that all communications have ceased and the connection can be closed. This alert prevents an adversary from performing a *truncation attack* to fool the server or client into thinking all the data to be exchanged have been sent, when actually there is some left (this can be a problem in situations such as banking transactions, where it is necessary for *all* the information to be received).

A.7 SSL Standards

SSL v. 3 is described in an internal Netscape, Inc. document. The TLS standard is described in IETF's RFC 2246. These documents only provide the basis for the SSL protocol—they do not specify exactly how SSL implementations should communicate. For this reason, the computer-security industry (including IETF, RSA Security, among others) has developed guidelines that are to be used by SSL implementors. This section briefly covers the SSL protocol and some of the more important standards.

A.7.1 Certificates

Abstract Syntax Notation 1 (ASN.1) was developed to provide a description language for telecom protocols. ASN.1 is a fairly complex language, and its details are not important, except to note that SSL uses ASN.1 to encode certificates in a computer-readable format. As a result, most implementations (exceptions include servers without certificate verification, which is described below) will need to be able to parse ASN.1. See Appendix B. "SSL Certificates," for more information on certificates.

A.7.2 Public-Key Cryptography Standards

In addition to the standards for certificates, RSA Security has developed a set of standards for public-key cryptography for use with TLS or SSL. These standards are (appropriately) called the *Public-Key Cryptography Standards* (PKCS). These standards apply when doing public-key cryptography operations, and are used extensively in SSL implementations.

There are currently about 13 PKCS standards (see www.rsasecurity.com/rsalabs/pkcs/) applying to RSA-based cryptography, Diffie-Hellman, Elliptic-Curve cryptography, and several others.

The standard most relevant to SSL is PKCS #1, the RSA Cryptography Standard. PKCS #1 describes the RSA algorithm, as well as versions of the algorithm applicable to implementation. Most importantly, it describes how *padding* is added to a message that is to be encrypted/decrypted using RSA. Padding is the name given to the extra (usually random) data added to a message to conceal its length, or to be compatible with a given cryptographic algorithm. Concealing the length protects an attacker from inferring secret information about the message, and some algorithms (such as RSA) require data to be a certain length.

PKCS #1 defines two padding algorithms, PKCS1-v1.5 and OAEP. The details of these algorithms are not important for this manual, but they both essentially add a number of random bytes (with special terminator characters) to the data before it is encrypted. All SSL implementations must include at least PKCS1-v1.5 to be compatible (OAEP is a newer standard that may have some security issues, so it has not become widely accepted).

A.8 SSL and TCP/IP

The most common SSL communication channel is TCP/IP. TCP is reliable, and so satisfies the in-order data requirements of the SSLv3 specification and the TLS RFC. SSL is a transport-layer protocol, designed to fit invisibly between the transport and application layers of the TCP/IP protocol stack. This section describes how SSL interacts with TCP/IP.

A.8.1 SSL, Sockets, and the TCP/IP Stack

Most SSL implementations are created with an API that is very similar to that of traditional network sockets. In fact, SSL is intended to replace sockets and require no other functional changes to add security to applications already using TCP/IP sockets. In theory, any application that uses sockets could be made secure just by replacing the socket API calls with SSL API calls.

Note that *communication* is different from *compliance*. An implementation without the mandatory features specified in the TLS RFC can still usually communicate with a TLS-compliant implementation, but is not *guaranteed* to be able to communicate with all implementations.

A.8.2 The SSL Record Layer

SSL is very similar to TCP in that it is designed to encapsulate data for communications. TCP provides routing information to get the data to its target, and SSL provides the security. Like TCP, SSL has its own message format, the SSL *Record*. All SSL messages are wrapped in SSL records, which contain the message type, the length of the message, and the version of SSL/TLS being used. The Record layer is independent of the underlying TCP message structure—a single record may consist of several TCP frames, and a single TCP frame may contain multiple SSL records.

A.9 SSL and HTTP

The intention of the original designers of SSL was to secure HTTP communications. The predominant high-level Web communication protocol, HTTP has no security whatsoever—it sends all data as plain text over the network. This makes applications such as e-commerce Web sites impossible to implement (no one would want to send their credit-card number over the Internet as plain text, where an attacker may be watching).

In theory, it is easy to integrate HTTP with SSL—just replace all the socket calls in the HTTP server with SSL calls. There are many problems with this because HTTP is a *stateless* protocol, that is, neither the server nor the client saves any information about the other during communication. However, SSL requires the state session information to be saved by both sides for session rengotiation.

A.9.1 HTTPS

The combination of HTTP and SSL is known as *HTTPS* (HTTP secure), and is described in IETF RFC 2818. Essentially, all HTTP data are encrypted and sent via SSL. When received, the data are decrypted and are then processed normally. However, the implementation of HTTP has to know that SSL is being used since all the data will arrive at once in an SSL record, rather than a line at a time as normal. (For example, some Web servers read a line at a time, waiting for a blank line to signify the HTTP headers have all been sent, which will not work with HTTPS.)

A.9.2 SSL and Web Browsers

All modern browsers support some form of SSL. Almost all new browsers support TLS. For this document, we will concentrate only on Microsoft Internet Explorer and Netscape, the most popular browsers.

The reason we mention browsers at all is that the implementation must be constructed carefully in order to be compatible with most browsers. Each browser's implementation of SSL differs slightly, and although all follow the specification, these differences can cause problems.

To illustrate the problem, let's look at the behavior of Internet Explorer and Netscape in a specific example. When the browser receives a certificate from the server, and verifies that it has been signed, but not by a trusted CA, the browser notifies the user and allows the user decide whether to cancel the connection or continue, even though the certificate is untrusted. This can happen often during debugging, or if SSL is used internally (all certificates are signed by the company, avoiding the CA fees). If the user chooses to continue, the SSL handshake should continue to establish the connection.

However, the SSL specification does not specify exactly how this happens. When a user chooses to continue with an untrusted certificate, the actions taken by Internet Explorer and Netscape differ significantly. Upon receiving the user's choice, Netscape continues the handshake and establishes the connection as normal. Internet Explorer, on the other hand, continues the handshake to completion upon receiving a "bad" certificate, immediately cancels the connection, and *then* prompts the user. It then does a session renegotiation to reestablish the connection if the user chooses to continue. This does not necessarily change the server's behavior, but it is a reason to implement session renegotiation (other than the performance enhancement).

The preceding example is just one of many problematic details in communicating with browsers. There are many other potential problems, such as when and how HTTP headers are sent, the timeout periods for different operations, as well as all the considerations for normal network communication.

A.9.3 Browsers and Certificates

Because Internet browsers are SSL clients, they must contend with managing and verifying certificates. They do this by storing several databases, one each for CA, intermediate CA (smaller corporations given signing authority by a major CA), Web site certificates, and local certificates. The browser uses these CA databases for verification, to store trusted certificates (Web sites), and to provide its identity (client certificates).

Appendix B. SSL Certificates

B.1 What is a Digital Certificate?

In order for security to be possible on the Internet, there needs to be some notion of trust. This trust is achieved by being able to determine the identity of some entity on the Internet with some degree of confidence—once this is possible, you can decide *who* to trust. A mechanism for identification on the Internet does exist, but how does it work? Most of the identification needed on the Internet for e-commerce and other purposes is provided by a single mechanism, the *digital certificate*.

A digital certificate is an electronic document that is used for identification purposes on computer networks (such as the Internet). It contains the name and (sometimes) contact information for the owner, along with other information used to identify the owner. The "owner" is typically the physical owner (an individual or organization) of a networked device.

Digital certificates would be useless if there was not some way to verify their validity. In the physical world, there are certain physical traits that are difficult to fake, such as fingerprints or DNA, and some of these can be used to determine whether or not a person matches the information on their identification document (such as a photo on a driver's license). In the digital world, however, there is no way to tell one person from another by physical traits. Anyone can claim to be the owner of a certificate. For this reason, digital certificates rely on what is called a *digital signature*.

The digital signature concept is based on public-key cryptography, which relies on separate public and private keys to encrypt a message. The public key is given out in the certificate itself, whereas the private key is kept secret by the owner. A digital signature is simply a message digest hash of the entire certificate (using either of the popular hashing algorithms, SHA-1 and MD5) that is encrypted using the owners *private* key. Normally, messages are encrypted using the public key, allowing only the owner of the private key to decode them. In reverse, however, the correct message hash can only be discerned using the public key to decode the digital signature. The way this works for validation is that the only person who could have encrypted the hash is the owner of the matching private key. Since the private key is kept secret and is difficult to duplicate then we can be sure with relatively high confidence that a digital certificate belongs to the owner of the private key. The problem is, there is no guarantee in the certificate itself that will tell you that the owner of the private key is who they say they are.

In the physical world there are certain measures taken to assure the validity of documents used for identification purposes, such as the holographic images on some drivers' licenses. These verification mechanisms usually rely upon the use of hard-to-duplicate items (such as holograms) so that the chances of the document being faked are low. These items are controlled by some organization (such as the Department of Motor Vehicles) and they provide a centralized service that does the work necessary to vouch for an individual. If you trust the organization, then you can be relatively certain that anyone possessing a identification document provided by the organization is who they say they are. This concept of centralized trust allows two people who have never met and know no one in common to assure their mutual identities (such as a person opening a bank account, the person has a DMV-issued identification card and the government assures the validity of the bank). The concept can be further extended to form a "chain of trust." In the case of the DMV, it is controlled by the government. Therefore, in theory, if you trust the government, you can trust the DMV, and therefore you can trust the documents produced by the DMV, forming a "chain" from the government to individuals.

This "chain of trust" is exactly the concept used by SSL for authentication in computer networks. On the Internet, the governing bodies for identification are corporations (such as Verisign and Thwate, the most recognizable of these companies) called *certificate authorities (CA)*. A CA provides the service of verifying the identity of a certificate owner using conventional means, then signs that owner's certificate using their own private key. The CA also has a certificate, called a *root certificate*, associated with the private key used for signing, that is publicly available for checking the authenticity of certificates. If you trust Verisign, then you can trust any certificate signed by them. Sometimes, a CA will issue secondary CA certificates, which means a third party can be responsible for signing certificates, and since that party has been authenticated by Verisign, you can then trust certificates signed by that party. In fact, anyone can be a CA, if they want to. All you need is a certificate, and the ability to sign other certificates using your private key. Anyone who trusts you can then trust the certificates you sign.

Certificates can also be *self-signed*. This means that the certificate is signed using its own paired private key. All CA root certificates are self-signed, and self-signed certificates can be useful for testing purposes, or in situations where authentication is not important.

Obviously, the entire digital certificate system relies on the "chains of trust". Without them, there would be almost no way to identify anyone for certain. Anyone could pose as anyone else. To prevent this, commercial CAs publish their certificates in popular Web browsers and Web servers. If you create your own CA, you will need to distribute your master certificate yourself, unless you are willing to pay Verisign to accept you as a third-party CA, or pay to have your certificate added to a browser (both are too expensive for most individuals and small companies).

B.2 What's in a Certificate?

Digital Certificates are encoded using the *Distinguished Encoding Rules* (DER) subset of the *Abstract Syntax Notation* (ASN.1) developed by the telecommunications industry in the late 1970's. This flexible format allows some fields to be omitted, and some custom fields to be created.

A certificate can be organized into sections[i] as follows:

- The owner's identification, usually a name and other information such as the location of the owner
- The issuer's information, usually a reference to the signer's certificate
- Validation information, such as dates when valid, serial number, and accepted uses
- The owner's public key
- The digital signature
- Custom sections and extensions

Each section serves a specific purpose, and is explained in further detail below.

B.2.1 Owner's Identification

This section of the certificate contains the identification information for the certificate owner. Sometimes identified as the subject of the certificate, this section contains specific fields used for identification. Only one of these fields is mandatory, the *Common Name*, abbreviated CN. The Common Name identifies the certificate with a particular network address, usually either a URL or IP address. Web browsers will compare this field with the domain providing the certificate and issue a warning if the two do not match.

i. Note that this breakdown does not match the actual fields in a digital certificate. However, this organization helps to clarify the functions of each of the different fields.

This section can contain a number of other fields as well, some of the more common being Country (C), Organization (O), Organizational Unit (OU), and Location (L). A couple of these fields, along with the Common Name, are usually all that is needed in the identification section. It is also possible to define custom fields for additional information, if needed.

B.2.2 Issuer's Information

The issuer's information is simply the issuer's root CA certificate identification section. On a self-signed certificate, this section is identical to the identification section. On a certificate signed by a CA, the issuer's information matches the root CA certificate's identification section, allowing applications to find the matching root CA certificate in their local repository to do the authentication check.

B.2.3 Validation Information

The validation information consists of a few items that help verify and manage certificates. These include a date range in which the certificate is valid (usually a start date, before which the certificate is invalid, and end date, after which the certificate is also invalid), a version number, and a serial number.

This section also includes a constraints field that indicates the uses for which the certificate is valid. These may include server authentication and e-mail authentication, as well as other possible uses for which the certificate may be valid.

B.2.4 The Public Key

This section simply consists of a field indicating the public-key algorithm used and the size of the key, and the public key itself. Most certificates today use the RSA public-key algorithm with key sizes ranging from 512 bits to 2048 bits.

B.2.5 The Digital Signature

Like the public key, the digital signature consists of a field indicating the algorithm used (usually either SHA-1 or MD5), and the signature itself. The signature is simply an encrypted hash of the entire certificate which can be decrypted using the public key. The signature is used to verify that the certificate has not been tampered with.

B.2.6 Custom Sections and Extensions

The flexible nature the DER format allows for custom sections to be added to certificates. These sections usually provide additional information not included in the identification section. Some examples include a "friendly name" (a more informative name than the Common Name) and application-specific *extensions* (an extension is simply a custom section specifically targeted at a particular application, such as a particular brand of Web browser). Extensions can hold any information needed by those applications.

Appendix C. API Functions

The following API function calls are for CGI programs. With the addition of HTTPS to the HTTP server, the socket is no longer accessible directly (although it is still accessible in the HTTP-only mode), since each HTTP server may be either a plain HTTP server or a secured HTTPS server. For this reason, we provide the following API function calls to CGI programmers, which allow basic direct-socket access using the current HTTP server. These function calls determine the current mode of the passed-in server structure, and use the appropriate socket (TCP or SSL) for that server. It is recommended that these function calls be used instead of direct socket access for new development.

http_sock_mode

```
void http_sock_mode( HttpState* state, http_sock_mode_t mode );
```

DESCRIPTION

HTTP socket wrapper function for socket mode. This function can be used by CGI applications to set the mode of a socket associated with a particular HTTP server.

PARAMETERS

state	HTTP state pointer, as provided in the first parameter to the CGI function.
mode	HTTP mode to use for the socket. Valid values for `mode` are: • `HTTP_MODE_ASCII` - Sets the associated socket to ASCII mode. • `HTTP_MODE_BINARY` - Sets the associated socket to BINARY.

RETURN VALUE

None

LIBRARY

`HTTP.LIB`

`http_sock_bytesready`

```
int http_sock_bytesready( HttpState *state );
```

DESCRIPTION

HTTP wrapper function for `sock_bytesready()`. This function may be used by CGI applications to determine if there is data waiting on the socket associated with a particular HTTP server.

PARAMETERS

`state`	HTTP state pointer, as provided in the first parameter to the CGI function.

RETURN VALUE

`-1`: no bytes waiting to be read
`0`: in ASCII mode, a blank line is waiting to be read,
or, for UDP, an empty datagram is waiting to be read
`>0`: number of bytes waiting to be read

LIBRARY

HTTP.LIB

http_sock_gets

```
int http_sock_gets(HttpState *state, byte *dp, int len);
```

DESCRIPTION

HTTP wrapper function for `sock_gets()`. This function can be used by CGI applications to retrieve a string waiting on an ASCII-mode socket associated with a particular HTTP server.

PARAMETERS

`state`	HTTP state pointer, as provided in the first parameter to the CGI function.
`dp`	Pointer to return buffer
`len`	Maximum size of return buffer

RETURN VALUE

`0`: if buffer is empty, or
if no "`\r`" or "`\n`" is read, but buffer had room *and* the connection can get more data!
`>0`: is the length of the string
`-1`: error

LIBRARY

`HTTP.LIB`

SEE ALSO

`http_sock_mode`, `sock_gets`

`http_sock_fastread`

```
int http_sock_fastread( HttpState *state, byte *dp, int len );
```

DESCRIPTION

HTTP wrapper function for `sock_fastread()`, that is for non-blocking reads (root). This function can be used to read data from a socket associated with a particular HTTP server. This function is intended for use in CGI applications.

PARAMETERS

`state`	HTTP state pointer, as provided in the first parameter to the CGI function.
`dp`	Pointer to return buffer
`len`	Maximum size of return buffer

RETURN VALUE

`>0`: the number of bytes read
`-1`: error

LIBRARY

`HTTP.LIB`

http_sock_fastwrite

```
int http_sock_fastwrite( HttpState *state, byte *dp, int len );
```

DESCRIPTION

HTTP wrapper function for `sock_fastwrite()`, that is, for non-blocking writes. This function can be used to write data from a root buffer to a socket associated with a particular HTTP server. This function is intended for use in CGI applications.

PARAMETERS

state	HTTP state pointer, as provided in the first parameter to the CGI function.
dp	Pointer to buffer containing data to be written.
len	Maximum number of bytes to write to the socket.

RETURN VALUE

`>0`: the number of bytes written
`-1`: error

LIBRARY

`HTTP.LIB`

http_sock_xfastwrite

`int http_sock_xfastwrite( HttpState *state, long dp, long len);`

DESCRIPTION

HTTP wrapper function for `sock_xfastwrite()` for non-blocking writes. This function can be used to write the contents of an xmem buffer to a socket associated with a particular HTTP server.

PARAMETERS

`state`	HTTP state pointer, as provided in the first parameter to the CGI function.
`dp`	Buffer containing data to be written, as an xmem address obtained from, for example, `xalloc()`.
`len`	Maximum number of bytes to write to the socket.

RETURN VALUE

`>0`: the number of bytes written
`-1`: error

LIBRARY

`HTTP.LIB`