The founder of Agranat Systems examines the design issues involved in engineering effective Web technologies for embedded systems.

Web browsers have become the de facto standard user interface to a variety of applications. They can run on almost any platform—from PCs and workstations to PDAs, cell phones, and pagers—and allow end-users to access Web-enabled applications from any location. Applied to embedded systems, Web technologies offer graphical user interfaces that are user friendly, inexpensive, cross-platform, and network-ready. System designers are therefore using them to supplement or replace traditional proprietary command-line and graphical user interfaces. Web interfaces offer benefits to manufacturers as well as to end-users. Because Web-based interfaces are cross-platform and easier to develop than traditional proprietary interfaces, manufacturers enjoy a shorter time-to-market for new product features. Moreover, the costs associated with product support, training, and documentation are lower, and the Web interface can be used for remote diagnostics of products in the field.

WEB TECHNOLOGIES IN EMBEDDED SYSTEMS
As with the Web in general, Web pages from the embedded system are transmitted to the Web browser, which implements the user interface. Some of these pages are unchanging and can be stored in the embedded system. In other cases, the embedded system dynamically generates the pages to convey the current device state to the user. End-users can also use Web browsers to send information to the embedded system for configuration and control of the device.

Web-enabled devices use the HTTP standard protocol to transmit Web pages from the embedded system to the Web browser, and to transmit HTML form data from the browser back to the device. The devices require a network interface such as Ethernet or RS-232, TCP/IP software, embed-
ded Web server software, and the Web pages (both static and generated) that make up the device-specific GUI.

Figure 1 shows a typical Web browser interface to a Web-enabled device with an embedded system.

**DESIGN ISSUES FOR EMBEDDED WEB SERVERS**

Traditional Web servers are designed to serve static Web pages from high-end workstations with plentiful CPU and memory resources. Embedded Web servers have different requirements for which traditional technologies are unsuitable.

**Protocol Considerations**

The core of any embedded Web server technology is the HTTP engine. HTTP has evolved rapidly from the simple HTTP 0.9 used in the first experimental Web servers, to HTTP 1.0, followed by the Internet Proposed Standard HTTP 1.1, which will soon be revised to an Internet Draft Standard. While implementing a subset of HTTP 1.0 is fairly easy, it was not accepted by the Internet Engineering Standards Group as an official standard for reasons that are critical in embedded applications.

First, HTTP 1.0 lacks explicit cache control. Browsers and intermediate proxies may cache Web pages under certain conditions (see Figure 2). For static Web pages such as graphical icons and logos, caching is desirable as it reduces the CPU load of the embedded system by eliminating requests for redundant information. However, dynamically generated Web pages must not be cached. Caching such pages would prevent the Web browser from retrieving up-to-date information about the status of the embedded system, thus rendering the Web interface useless. HTTP 1.0 Web servers in embedded systems can cause misleading information to be delivered to end-users. They can also experience excessive load by serving redundant information. HTTP 1.1 has corrected this problem by adding explicit cache control mechanisms.

A second major problem with HTTP 1.0 is that each HTTP request requires its own TCP connection. A typical Web page may consist of several individual HTTP requests: one for the base page, one for each HTML frame, and one for each graphical image. Establishing and tearing down each TCP connection requires significant CPU and memory resources, especially since TCP implementations must maintain connection state information for two minutes after the connection has been closed. HTTP 1.1 allows for a single persistent TCP connection between the browser and the server for multiple HTTP transactions, thus greatly improving network and system performance (see Figure 3).
The correction of these two problems in HTTP 1.1 gives embedded Web servers a reliable user interface with minimal impact on the memory and CPU resources of the embedded system.

**Embedded Software Considerations**

Embedded systems have limited CPU and memory resources available to them. In many cases, these limited resources are largely committed to mission-critical and real-time applications. For example, a data communications router may require most of the CPU and memory resources of the embedded system to forward packets. Unless the embedded Web server software is sensitive to the real-time requirements of mission-critical applications, it might drop packets or lose data. In fact, the embedded Web server is often the lowest priority service in the system, so end-users can wait hundreds of milliseconds for a response (an eternity compared to the low-latency requirements of many embedded applications).

Traditional Web server software designed for workstations typically requires hundreds of kilobytes of code and megabytes of additional runtime memory. In contrast, well-designed HTTP 1.1 Web servers can require as few as 10 K bytes of memory. Also available is embedded TCP/IP software that requires only a fraction of the memory used by most workstation implementations. Efficient TCP/IP and Web server software can add as little as 48K ROM and 16K RAM to the memory requirements of a Web-enabled embedded system.

Few embedded systems have an operating system. Therefore, to be portable to any embedded software environment, embedded Web server software should run independently of operating system services, such as multitreading. At the same time, for optimal user interface performance, the embedded Web server software must be able to respond to multiple and simultaneous HTTP requests, especially when using older HTTP 1.0 browsers.

Finally, embedded Web server software should use nonblocking interfaces to the embedded application and the TCP/IP stack such that one HTTP transaction waiting for data will not prevent another HTTP transaction from accessing the device.

**Embedded Application Interface Considerations**

Embedded Web server software must provide mechanisms for the embedded application to generate and serve Web pages to the browser and to process HTML form data submitted by the browser. Designing the interface between the embedded Web server and the embedded application offers the greatest challenge.

**Common Gateway Interface.** One possible solution is modeled after the CGI found in many traditional Web servers. In this model, each URL is mapped to a CGI script that generates the Web page. In a typical embedded system, the script could then send raw HTML, XML, or other types of data to the browser by using an interface provided by the embedded Web server software. Figure 4 presents an example CGI script for an embedded system used to operate a refrigerator thermostat.

While this approach is perhaps the easiest for the embedded Web server developer, it is by
far the most difficult for the GUI designer. CGI scripts are tedious to write. Once written, the “look and feel” of the Web page can only be determined when the script is executed. In an embedded system, this implies building the executable image, burning it into Flash, and booting the device before the Web page can be viewed by a browser. Therefore, CGI solutions slow the time to market and are difficult to maintain.

Server-side scripting. Another solution is to use server-side scripting. With this approach, Web pages are first developed and prototyped using conventional Web authoring tools and browsers. Next, proprietary markup tags that define server-side scripts are inserted into the Web pages. (Figure 5 shows the server-side script in a sample HTML page.) The marked-up Web pages are then stored in the device by using a simple file system. When a marked-up Web page is served, the embedded Web server interprets and executes the script to interface with the embedded application. For example, a proprietary scripting language could define an interface to invoke application functions used to generate the dynamic part of a Web page. In Figure 6, embedded system initialization code is used to register

---

Figure 4. A CGI script running in an embedded device provides the internal application interface used to generate Web pages on the fly.

Figure 5. Example of HTML page containing a server-side script.

Figure 6. In an embedded system, initialization code is used to register the server-side scripts with function calls.
the server-side scripts with function calls. The code for a function call invoked when server-side scripts are interpreted by the server is shown in Figure 7.

Server-side scripts are easier to use than the raw interfaces of CGI solutions. However, interpreting scripts at runtime in an embedded system may impact system performance. Moreover, a significant amount of memory is required to maintain a database for mapping script names into embedded software functions and variables. Server-side scripts generally offer limited capabilities, resulting in larger and more complex embedded application code.

The HTML-to-C Preprocessor Approach. A third solution makes use of a preprocessor tool that converts Web pages into C code. Like server-side scripting, this approach uses conventional Web authoring tools and browsers to support quick development and prototyping of Web pages. The Web pages are then enhanced with proprietary markup tags that encapsulate fragments of C source code. These code fragments provide simple and efficient interfaces to the embedded application software. Figure 8 is an example of an HTML page containing encapsulated source code.

The preprocessor tool compiles the Web pages, strips out the proprietary tags, parses HTML forms, and generates C code. The C code is then compiled and linked with the embedded Web server and application software to produce a tightly integrated executable image (see Figure 9). The preprocessor enables sophisticated dynamic Web-page capabilities by performing complex tasks up front and generating an efficient and tightly integrated representation of the Web pages and interfaces in the embedded system. Figure 10 is an example of form interface code.

The HTML-to-C preprocessor offloads substantial Web server processing from the embedded system. Furthermore, the embedded C code and form-parsing offer the greatest flexibility for developing embedded application code. This approach supports a small, efficient embedded Web server that increases system performance.
CONCLUSIONS

Small embedded TCP/IP stacks and Web server software now make it possible to manufacture reliable, inexpensive Web-enabled devices across many industries and markets. Embedded systems require Web servers that are designed to minimize memory footprint and avoid interference with mission-critical and real-time applications. To guarantee a reliable user interface with minimal impact on system performance, the server software should utilize the latest HTTP 1.1 standards from the Internet Engineering Task Force.

It won't be long before intelligent devices worldwide will be nodes on a network and managed from Web browsers.

Ian Douglas Agranat is founder and CEO of Agranat Systems Inc. Agranat Systems provides embedded Web technologies to manufacturers of microprocessor-based products. The company is a member of the World Wide Web Consortium and contributes regularly to the Web standardization efforts of the Internet Engineering Task Force. For more information, visit http://www.agranat.com/.

Figure 9. The preprocessor compresses the Web pages, strips out the proprietary tags, parses HTML forms, and generates C code. The C-compiler then compiles and links the code with the embedded Web server and application software to produce a tightly integrated executable image.

```c
void cFormServe_temp ( CONTEXT ctxt, cForm_temp * form )
{
    form->value.temp = Temperature;
    form->status.temp = INITIALIZED;
}

STATUS cFormSubmit_temp ( CONTEXT ctxt, cForm_temp * form)
{
    if (form->status.temp & RETURNED)
    {
        Temperature = form->value.temp;
        return OK;
    }
    else
    {
        return ERROR;
    }
}
```

Figure 10. Example of form interface code generated by the preprocessor tool.

URLs for this report

Agranat • www.agranat.com/
HTTP 1.0 • ftp://ftp.isi.edu/in-notes/rfc1945.txt
HTTP 1.1 • ftp://ftp.isi.edu/in-notes/rfc2068.txt
IETF • www.ietf.org/