Embedded Web Servers as User Interface

ESC-343 • Mark Bereit

October 28, 2008

This paper talks about embedded devices offering a user interface that is presented within a standard web browser. That is, the means (or one means) by which the user of some device configures, administers, or checks the status of their embedded product is by browsing to the device as if it were some web page.

The web browser user interface is becoming a popular approach in embedded devices. This popularity is driven by two simple factors: more and more embedded devices are attached to a network, and pretty much everyone has a web browser. This allows devices to replace, or at least supplement, configuration interfaces of DIP switches or buttons and LCDs or similar approaches with some embedded code that can have low development cost and virtually zero production cost and—remarkably enough—make the end user happier.

It all sounds pretty good, and it can be... although anyone who has worked in the world of embedded would be justified to view with suspicion any sentence like the previous one that combines the phrases "some embedded code" and "low development cost." I want to talk about embedded web servers as a user interface, and provide some useful background for getting started without getting too complex or too expensive.

In this paper I'll talk first about the business case for the web browser user interface: where it works and where it doesn't. Then I'll talk about the basics of how web browsers and web servers work, focusing on the parts you need to care about in building a web user interface. Next I'll talk about some of the more recent extensions to the web browser that can make your user interface more polished (but also more technically complicated inside)... which you may or may not care to use, depending upon your market and your technical constraints. And then I'll talk about tricks to get the most out of tight constraints in terms of code or memory space and bandwidth, since many embedded devices don't have lots of room to spare.

That's a lot to cover in one paper... so let's get started!

When to use a web browser user interface

The short form of when to use a web browser as your user interface can be stated very simply: whenever it will make sense to your user. You're proposing that when the user needs to do some thing, you want him or her to reach for a web browser. Ask the question: will the user find this convenient, or consider it a nuisance? (Compared to other ways the user might do that same thing, of course.)

Part of this comes from the question of how frequently, and under what circumstances, the user will do the thing you're talking about. Are you talking about something the user does multiple times a day, or something that is pretty much done once and left alone? If it's a commonly recurring task, then a web interface only makes sense if your user is already at a computer when this task needs to happen. If it's an infrequent setup task, then the need to go to a web browser can be more defensible as long as the typical customer is sufficiently comfortable using one.

For relatively simple devices, I would suggest that it is rarely sensible to add network connectivity to a product for the sole purpose of giving it a web user interface. If the device doesn't have a reason to connect to a network already, a web interface probably won't make sense to the user even if it does work out cheaper to throw in an Ethernet adapter and some code than some control buttons and a display. But... if the user interface is inherently complex (the product displays a lot of complex information, and/or requires a lot of complex setup or monitoring), then using a computer may make more sense than a complex (and expensive) screen and buttons or touch interface. This is why an increasing amount of test and diagnostic equipment is using a computer interface where a stand-alone, complex (and expensive) machine might have been used in the past.

The product design part of the decision really is simple. Round up a few prospective users and tell them your product could use a web browser to do certain tasks, and see which way they respond: "great!" or "why?"

The accounting part of the decision always comes back to cost justification. If the product already has a network adapter (basically, Ethernet and/or WiFi) and a TCP/IP protocol stack, then adding an embedded web server may cost no more in production than some additional code space on the microcontroller (usually pretty cheap, although this depends upon how much space you can afford), and may well eliminate the cost of a more physical control interface (say, pushbuttons and a small display panel). For many products the largest cost of the web interface over the life of the product is actually the time spent writing the code... and hopefully this paper will demonstrate that this doesn't necessarily have to be hard.

OK: let's assume that it seems to make sense for your widget to use a web browser for its user interface, for some combination of the usual reasons (it will benefit the customer, it will save your company money, and/or someone promised it to a key account and now you have to deliver). This means that you have to build some sort of embedded web server into your product. Let's look at what this means.

Web browser / web server basics

The display of information in a web browser rests on two simple technologies that you will need to care about (along with a number of other technologies which you *may* also care about... but which we can ignore until later in this paper). These two central technologies are HTML and HTTP. (Not coincidentally, the first two references at the end of this paper are for a good book on each.)

HTML is a particular file layout that describes to a web browser what information to display, and how to present it. This controls what the web page looks like.

HTTP is a web protocol by which a web page is requested from some server in order to be displayed. This controls getting the page in the first place.

I'll talk more about each of these technologies in a moment, but let's quickly look at the role each plays. You have a web browser, passively displaying your home page or wherever you were last. In the address bar, you type a web address (called a URL) for where you want to go. The web browser (most likely) uses HTTP to go ask some server somewhere for the web page that corresponds to that address. That server gives your browser some data that is (most likely) an HTML file. Your browser renders the HTML file as a display on your screen, according to a combination of what the HTML file says, your browser's own rules and your preferences. And then you have a new page on your browser. As you know, a web page may contain links (typically rendered as underlined text, or some sort of button) that take you to other pages; the same process of HTTP to fetch and HTML to describe is used for each.

For simple reference pages on simple Internet web sites, this is all pretty static, where the HTML page you get back represents a file somebody posted at some previous time. Creating a user interface within a web browser, at its heart, requires only two minor refinements on this process:

- The data from the server may be generated on the fly, representing the status of the moment rather than some static content.
- A particular HTTP request may cause some effect, beyond simply requesting the next page of content.

That's the whole embedded web server story in a nutshell: standard web page service, with the ability to generate some content dynamically and/or react to certain requests.

HTML: describing a web page

Looking at the Internet today, with its vast array of content (some of which is even useful), with graphics and video and sounds and colors and excesses of bad manners, poor taste, offensive content and endless advertising, it may be difficult to remember that the point of the World Wide Web was to make it easy to browse through related research papers. But, there you have it. At its heart, HTML is a way to describe simple text documents with cross-reference links.

HTML stands for "Hyper-Text Markup Language." The "hyper-text" term refers simply to the idea of embedded cross-references, so that you can jump to a reference at the moment you read it (and jump back when done) rather than having to look it up later. The "markup language" means that HTML starts out as plain old boring *text*, like something you could write in Notepad, but the text is then "marked up" with additional information, such as instructions to make a phrase bold or italicized, or treat some text as a title, or arrange some text in a table, or begin a new paragraph at a certain point, and so on. A basic fact of any markup language is that if you stripped away all of the markup, what you have left is simple text. Text with all formatting and paragraph divisions removed may be less readable, but the information is still there in its basic form.

One key fact to take away from this: it is extremely easy to make boring HTML content; just don't do any markup! Conversely, the more you want to make your content pretty and graphical with precise visual layout, the more markup work you have to do (and the larger the HTML content becomes). We'll talk more about this trade-off later.

Consider the following example of a simple HTML file, and how it is displayed on a web browser.

```
<html>
 <head>
  <title>Writing assignment 1</title>
 </head>
 <body>
  <h1>What I Did For My Summer Vacation</h1>
  <p>Why do we get these <em>stupid</em> writing assignments, anyway?
   All I did this summer was play
   <a href="http://www.miniclip.com/">computer games</a>.</p>
  <p>For more great essays, <a href="essays.html">click here</a>.</p>
 </body>
</html>
```

In HTML, the markup occurs within “tags,” recognizable by the angle brackets (“<” and “>”) surrounding them. There are typically matched beginning and ending tags with the same tag name, where the ending tag is prefixed by a slash (“/”). (There are standards for when you don’t need the ending tags, and further standards for why you should have them anyway; see a good *recent* book on the subject.)

In my example, the outermost tag (“html”) simply marks everything as HTML content. The “head” tag indicates a header section of the file, information that is for the browser, distinct from the “body” tag marking information for the viewer. The “title” tag indicates the page title as displayed on the title bar of the web browser. The “h1” tag indicates a first-level heading (typically displayed in some large and bold text), each “p” tag indicates a paragraph, the “em” tag indicates emphasis (typically rendered as italics), and each “a” tag indicates a hyperlink to another web page.

Take a look at the two hyperlink tags. Each has an *attribute*, “href,” that tells the web browser the address of the page targeted by the link. The first one is a full web address, like you might type in the address bar of a browser, while the second is just a file name, interpreted by the browser to mean it lives on the same server and in the same folder as the current page. I’ll talk more about web addresses shortly.

If you strip away all of the tags, you are left with just the title and some text. That leads to an important concept in HTML: if the web browser doesn’t recognize a tag, it simply ignores it. For example, earlier versions of the HTML standard didn’t include the “em” tag (they used the “i” tag for italics instead), so an old web browser wouldn’t know to italicize the word “stupid.” But you’ll notice that failing to apply the rendering doesn’t hurt the document much.

I’m not going to spend more time talking about HTML here; get a good book on the subject. My personal favorite is *HTML & XHTML: The Definitive Guide*, sixth edition, by Chuck Musciano and Bill Kennedy and published by O’Reilly. The important points for now are that HTML really is a syntax for decorating text, and that it isn’t hard to learn the basics. (Of course, a good user interface probably won’t be satisfied with pages that *look* basic; we’ll get into the fancier stuff later.)

HTTP: retrieving web pages

You type a web address into your web browser... say, “www.google.com.” You press Enter. The Google page magically appears. The page received by your browser is HTML, telling the browser where to put the Google logo and the search box and all the rest of the things. But the mechanism by which your web browser *got* that HTML page from your web address involves HTTP.

HTTP stands for "Hyper-Text Transport Protocol," because it is a network protocol for transferring (mostly receiving) hyper-text documents. Except that it doesn't have to be hyper-text. When you download an MP3 file from a web site—legally, I'm sure—you're probably still using HTTP to get it. And when a web page includes graphics or videos or advertisements, these probably all came to you through HTTP as well. So at its most basic, HTTP is simply a way of asking the network to give you some public file from some network address.

Let's talk about network addresses for a moment. The fancy name for a web address is a URL, which stands for "Uniform Resource Locator." That simply means there is a standardized way that web addresses are formed to refer unambiguously to a particular bit of content. Here's a complicated example:

```
http://www.myserver.com:8000/user/scripts/lookup.php?q=Mark+Bereit
```

This might be a request for some custom search engine to look up this paper's author, perhaps to see how long it takes him to get to the point.

The first part—"http://"—tells the web browser to use HTTP to get the thing. That might seem unnecessary, but in fact web browsers have several ways to request information. For example, a web browser can fetch content from a file ("file://"), it can fetch content using FTP (an older file transfer protocol; "ftp://"), and it can also fetch from a *secure* version of HTTP ("https://"). In the absence of this part, a browser will assume that you want to use HTTP and fill that in for you.

The second part—"www.myserver.com"—is the name of the computer that the web browser has to talk to. This is any valid computer name, and can equally be a network "IP address" like "64.233.169.104" instead... which is important for us, because often embedded devices only have an IP address, and no name to look up (or, more to the point, your computer doesn't know who to ask to find out how to reach your embedded device by name).

The third part—":8000"—is rarely used. It tells the web browser a particular "port" number for the network connection to use when it talks to the computer. When this is omitted, your web browser uses a standard port number without being told. (The defaults are port 80 for HTTP and port 443 for HTTPS.)

The fourth part—"/user/scripts/"—is the folder containing the particular page file, if any. This is the same idea as a file path on a computer (although the web uses the forward slash character, not the backward slash used in Windows). At a minimum, a path of "/" indicates that there are no folders between the "root" of the web server and the page you want.

The fifth part—"lookup.php"—is the particular "page" or other resource on that computer that you're asking for. If no page is requested, your web browser still asks the computer for a page with no name; web servers are almost always configured to return a useful home page in this case.

The last part—"?q=Mark+Bereit"—is optional parameters to the page; in this case, a parameter whose name is "q" and whose value is "Mark Bereit" (the "+" indicates a space character). If the page being accessed is not just static content but some sort of program or script or any sort of dynamic content, then any parameters are processed as part of the request.

The URL uses certain punctuation characters to mean particular things: slashes, colons, and question marks all have particular meanings to let the web browser pick apart the pieces. Unexpected punctuation characters within the parts could cause the browser or web server to misunderstand the URL, so a punctuation character within one of these parts is represented by a percent sign and two hexadecimal digits indicating the particular character you want. (A space character, however, is represented by the plus sign.)

Now, a web address that starts with the file name (say, "essays.html") is assumed by the web browser to mean that the current protocol, computer, port and any path are all still valid, so the browser just substitutes the new name at the end of the current path. Similarly, a web address that starts with the path (say, "/users/mark/essays.html") is assumed by the web browser to mean that the current protocol, computer and port are still valid, so it substitutes the new path at the end of the current port or computer. This is done by the browser, not by HTTP: every HTTP request has a protocol, computer and port, from which the browser requests the particular path, page and parameters.

So the web browser has all the information of a full URL. What it does next depends upon the protocol. In the case we care about, HTTP, the web browser opens an Internet connection from itself to the computer you specified, using the port you specified or the default HTTP port. It sends a simple request header, all in plain text, telling the computer a little bit about the requesting computer and the particular resource (path, file, parameters) it is asking for. The web server on that computer, having already been configured to listen for and accept requests on a particular network port, answers the request with a response header, again all in plain text, telling the browser a little bit about the computer and the results of the request... followed, we hope, by the requested page or file or resource. And then the server computer closes the connection.

If you were somehow able to peek inside this exchange (using a network packet sniffer program, for example), for the URL example above you might see this request go out from the web browser to port 8000 on www.myserver.com:

```
GET /user/scripts/lookup.php?q=Mark+Bereit HTTP/1.1
Host: www.myserver.com
Accept: *
Accept-Language: en-us
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; en-US; rv:1.9.0.1)
Gecko/2008070208 Firefox/3.0.1
```

And you might see this reply from the server:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 126

<html><head><title>Search Results</title></head>
<body>Mark Bereit is a reasonably OK guy, but talks too much.</body>
</html>
```

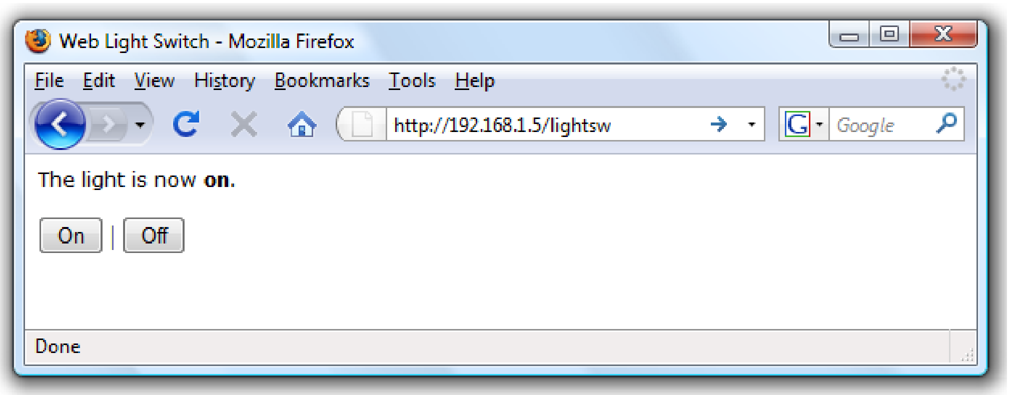
The web browser sees a response code of "200" that indicates success (the "OK" is just there for us humans), and knows that after the blank line at the end of the header, everything else in the response is the actual content requested, which it processes as HTML for display.

There is a lot more to HTTP than this, of course; again, a good book is important. I like *HTTP: The Definitive Guide*, by David Gourley and Brian Totty and published by O'Reilly. This exchange of information is ordinarily between the web browser and the web server and not your problem... unless, of course, you are writing some of the web server. Which, in embedded systems, I don't rule out.

A web page with functionality

So far we've just talked about fetching pages from the Internet, which isn't all that helpful for an embedded user interface. But let's move now to a simple example of a web page for an embedded device.

Suppose that we have a web page representing a light switch. (It doesn't get much simpler than that!) The page might look just like this:


Web Light Switch - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.1.5/lightsw
Google
On
Off
Done

The light is now **on**.

I'm not going to win any creativity awards for this page, but it is clear enough: it tells me the current status (important since, unlike a regular light switch, I may not be close enough to see whether the light is on) and clearly shows where I click to change things.

How does the page *know* whether the light is on? Well, this page is generated in code, not a static file. Somehow the embedded web server knows to run a certain chunk of code if the requested page is "lightsw." If the page were coming from C code, for example, there might be a line in there something like this:

```
cb += sprintf(szReturn+cb, "<p>The light is now <strong>%s"
    "</strong>.</p>", gbLight ? "on" : "off");
```

As for the "on" and "off" buttons, these are generated by a little chunk of HTML like this:

```
<form action="lightsw" method="post">
<p><input type="submit" name="switch" value="On" /> |
<input type="submit" name="switch" value="Off" /></p>
</form>
```

The "form" tag indicates a section of the page for the purpose of submitting user-provided information (like those forms that ask for your e-mail address in order to provide you with more spam). In this case, the form will be submitted right back to the same page (the "lightsw" page, which is already the page in the address bar above). Each "input" tag in this example, with a "type" attribute of "submit," indicates a button which, when pressed, sends all the parameters collected for the form to wherever the form goes. My example is very simple and there aren't any input fields on my "form," so the "form submission" goes to the server with just one piece of information, depending on which button I click, such as:

```
switch=Off
```

How does this information get back to our page? Well, my "form" tag told the browser to use the "post" method; this sends parameters as additional lines in the HTTP request. In this case, the HTTP request from the browser might look like this:

```
POST /lightsw HTTP/1.1
Host: 192.168.1.5
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; en-US; rv:1.9.0.1)
Gecko/2008070208 Firefox/3.0.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 11

switch=Off
```

Had the "form" tag specified the "get" method instead, the web browser would have added parameters to the URL, doing a regular page fetch from this URL:

```
http://192.168.1.5/lightsw?switch=Off
```

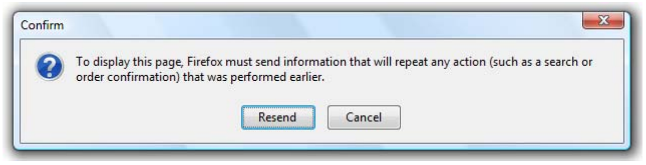
In either case the parameter is sent to the server, and code running on the web server must have some way to see the parameter and act upon it.

Now, it might seem easier in our situation to not bother with forms and parameters, but just have action pages: say, a dynamic page named "on" and a dynamic page named "off," with hyperlinks to each, to set the switch. Well, ignoring the fact that your user interface may need more functionality than just action commands like this, there are problems with such an approach, and the biggest of these is the **page cache** used in web browsers.

The idea of a page cache is, generally, a good thing: every time the browser fetches a page, it generally keeps a copy on your hard drive so if you ask for it again soon, it doesn't have to go out to the network to get it. This repeat use happens more often than you think. Remember that every fetch of a graphic or logo from a web site is its own HTTP transaction that takes time; the more of these your browser can recycle from a recent fetch, the faster your pages load. But this optimization gets in the way of embedded web pages: suppose that yesterday you used the "on" page to turn on your light, and that page is in your cache. Today you go to the "on" page again to once again turn on your light... but your web browser finds a copy of the page in its cache, so it displays that *without talking to your server*. Your server never hears that you want the light on... and worse, your web browser is displaying yesterday's page that said it did it.

There are a variety of tricks you can use to try to tell the browser to not cache the page, and there are a variety of Internet Service Providers and firewalls and utilities that try to aggressively hold onto content in the same way that your browser does that may undermine your efforts. It's quite a complicated subject. To sum up: good luck with that. On the other hand, your browser (and hopefully any intervening web cache) knows that requests with parameters should always go through to the server instead of drawing from a cached copy.

Your web browser also treats a "post" form differently than a "get" form. A "get" is like any other page, with parameters in the URL; if you are on such a page and click the browser's Refresh button, the browser submits the request again. A "post" is considered to have possibly changed something on the server, so if you click the Refresh button on such a page, the web browser asks whether you actually want to do this thing again. This is because HTTP defines a "get" as a "safe" operation that has no side effects, while a "put" may cause a change.


Confirm

To display this page, Firefox must send information that will repeat any action (such as a search or order confirmation) that was performed earlier.

You may have no problem with a refresh of the “on” or “off” page causing the light switch to be set that way again, or then again you might consider that to be a bad behavior. My example used the “post” method because using the “get” method doesn’t really match the HTTP expectation of no side effects.

The embedded web server

So far we’ve covered the core pieces of what you’re trying to do. You need to include in your embedded device some sort of web server that has the ability to hand off particular page requests to code. The code needs to be able to generate HTML content that may be affected by the state of the device. And the code needs to be able to access the “post” or “get” parameter(s) that came with the request in order to know exactly what it is being asked to do. (You may need to access other request header information, and/or specify response header information, as well.)

Exactly *how* you do this depends upon your particular embedded web server. Most firms that sell an embedded operating system with networking, or that sell a TCP/IP software stack, include or also sell a web server module. In the modern world of widely available network technology there is probably no good reason to write your own web server, and plenty of reasons not to: while basic web server code is not horribly complex, it can be tricky to fully debug, and (speaking from experience here!) the server can seem to run well for a while before you find another “gotcha” that makes it fail in the real world. Plus, basic code doesn’t scale well if there are lots of requests, and it probably can’t handle HTTPS (secure HTTP) when you find you need it. So, simply put, look to your TCP/IP code provider. (If you are not only writing your own web server but also your own TCP/IP stack, smack yourself right now. A project like that could be fun for hobby work, in which case *TCP/IP Lean* by Jeremy Bentham would help a lot... but don’t put your experiment into a product for real customers.)

The implementations of a particular web server vary widely. Larger embedded systems with 32-bit or 64-bit cores and hard drives usually have web server systems that are similar to, or copies of, the web server applications used on full-blown server PCs; these may launch dynamic code as separate processes or applications, often as scripts in high-level languages, as well as serve up static pages with no code intervention. (In larger Linux embedded systems, Apache is the preferred approach; in Windows embedded systems, IIS is the usual way to go.) Smaller embedded systems more likely handle the dynamic web content in-process, calling some dispatch mechanism you can override to generate content for a request, and serving static pages from the hard drive or from Flash memory or wherever your resources live.

I’m not going to talk about any particular web server or platform in this paper; details of my experiences may not fit your particular situation. I’m going to stick to topics that apply no matter what server code you’re working with. For your particular details, all I can advise is that you look into the documentation you have for whatever web server module you are using, and see how it does the basic things: serve up static content; serve up your dynamic content; let your dynamic code “see” the “post” or “get” parameters or other request header values; and let your code manipulate the response header values if needed. Try to match up their terminology with what I've talked about.

Even though the important part of the web server is serving dynamic content, you'll still need to serve up static content, too. There will be resources, and perhaps pages, where there is nothing dynamic to display. But it turns out that many dynamic pages are actually mostly static: only small pieces of the content ever change. (In my light switch example, only one word is ever different.) If your web server module provides a way for you to have pages that are mostly static with dynamic sections, take it. For example, Microsoft's IIS web server technology has "Active Server Pages," which are essentially static web page files with code embedded into them for the sections that are dynamic. If you have something like this it can be a great time saver in your development... and it also reduces your errors and enhances maintainability when you aren't constantly intermixing HTML syntax with the syntax of your programming language.

An address for your device

Your embedded device is going to need to have its own web address. At a minimum this means that it has some IP address by which people can reach it. This isn't a requirement of web servers in particular; this is probably a requirement for your embedded device to exist on a network at all. However, if you're going to have a customer access your device through his or her web browser, then the user will have to know the address for your device.

This can be more problematic than it first appears. Many networked devices can use DHCP to ask the network for an IP address, and accept whatever it is given. Well, that's fine, but then what do I type into the address bar of my web browser to talk to it? There may need to be some way that I can tell, from the device, what address it has received; this may bring back the need for a display or physical interface, diminishing the value of the web interface. On the other hand, you could simply set your embedded device for a particular default address and tell your customer what address to type... but the address you choose may conflict with an existing device on the network and/or be outside the customer PC's "subnet," meaning that typing the IP address still won't work for the browser (unless the customer goes to the added complications of reconfiguring the network adapter long enough to work with the IP address you chose... *not* something you want to lead most customers through in a manual or support call).

Depending upon the network environment, it may be reasonable for your embedded device to be identifiable by name to the network. If your device is working solely within a Windows network environment, for example, your device could not only use DHCP to get an IP address, but also register its own "machine name" with WINS—perhaps, a concatenation of the brand name and a serial number visible on a sticker on the device—so that the user can type the name, rather than the IP address, into the web browser.

You could provide a piece of software, perhaps part of a setup program, that "finds" your device on a network for a customer. (This software could communicate with the device using broadcast UDP, which can reach local devices on the LAN regardless of their IP settings, a trick I have used successfully for a number of products.) However, this limits your approach to particular types of computers that can run your software, instead of the wider audience of web browsers.

The design decision of how your device gets and/or announces its network address is not specifically related to embedded web user interfaces... but the need for the customer to type in the correct URL may influence that decision.

On a more positive note, whatever approach you take to solve the problem of finding your device's address, a web user interface is certainly a good way to *configure* network settings. The choices of IP address, subnet mask, gateway address, and perhaps other settings, are all handled easily in a form on a web page, and not handled easily or well with buttons and LCD displays.

Secure web communications

One thing to keep in mind about HTTP: it's wide open. I talked before about peeking at the communications between web browser and web server with a network packet sniffer. Well, that is a danger, if what is going back and forth is supposed to be private or secure.

Fortunately, the answer is really very simple, as long as your web server code module supports it: SSL. This stands for Secure Sockets Layer, which is a network protocol "wrapper" that uses encryption for the communications. Any web URL that starts with "https:" instead of "http:" tells the web browser to use a secure (though somewhat slower) web channel to otherwise do exactly the same HTTP things. And then the packet sniffer doesn't help anyone. The whole transaction (both sides) is encrypted and unreadable to anyone in the middle.

Graphics in your user interface

So far I've talked about the basics, and (in principle) you know how to go about making web content for your device. But you're certainly going to want the user interface on your product to look more polished and professional than my light switch example! So for the rest of this paper I'll be introducing more technologies and options you *may* want to explore to benefit your user interface... and the obvious place to start is with including graphics on your web pages.

Any decent book on HTTP will talk about how to do plenty of visual effects: not just placing graphics on your web page, but also colors and fills and font selections and the like. Although any decent book on HTTP will *also* tell you that the assistance of a graphic designer can be a very valuable thing, because without careful thought to what will look good and still be useful, you can with the best intentions create a web page that is a hideous thing to look at or use.

That said, of course you can put graphics in your pages: pretty buttons to click instead of the ones web servers offer by default; your corporate logo or product logo; graphics showing that light bulb from my example on or off instead of using text... whatever. Graphics are added to HTML through the "img" tag, which tells the browser to insert a graphic, such as this:

```
<img src="/img/companylogo.gif" width="150" height="50" border="0"
alt="Company logo" />
```

(This tag does not contain any text, and there is no ending tag; the slash just before the closing angle bracket stands in for an ending tag where none is needed.)

The "img" tag tells the browser about the graphic: its size, a text equivalent of the graphic, and the image file itself, which the browser gets from... well, from you. You have to serve up the images you use.

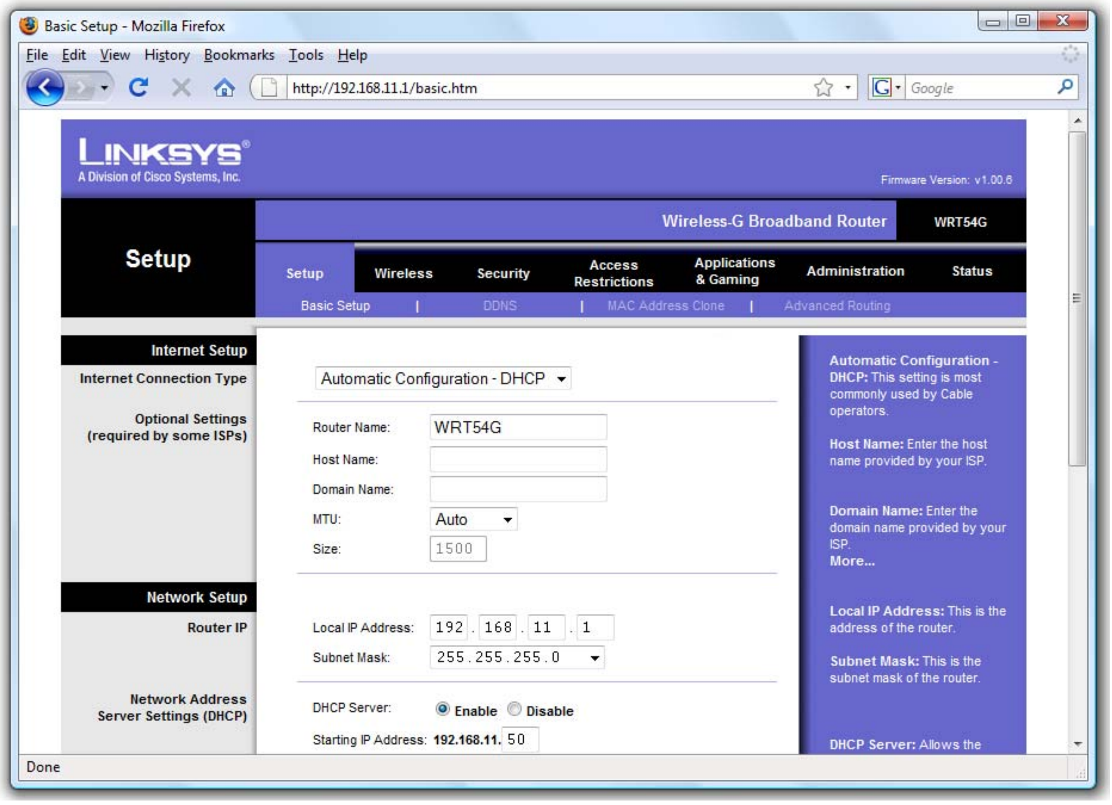
Well, technically speaking, you could have your "img" tag point to a graphic on the Internet, but now there's a problem: what if your customer's web browser can't reach the Internet? Instead of your intended graphics they see nothing, or little "broken link" indicators, and maybe the layout of the page is garbled because of the missing pieces. Trust me when I say that if this happens, the customer won't think to blame the Internet. They will blame your product for displaying something ugly and unusable.

So your product needs to contain, and serve up, any graphics you want to use. Keep this in mind when designing your user interface: the more graphics you use, the more you have to store in your system Flash memory or hard drive.

Web browsers universally accept graphics in the ".gif" format (good for icons and simple images of no more than 256 colors) and the ".jpg" ("JPEG") format (good for photographs and graphics where lots of colors are more important than sharpness of detail). These two formats should address most graphics needs.

Note that the web browser expects to get these resources tagged with the appropriate type in the HTTP response header: image/gif or image/jpeg. If this is not true, the web browser may handle them incorrectly. Your web server module may be smart enough to automatically apply these types based upon the file extension (full web server applications do this), or it may be your responsibility to tell the server the "MIME type" for any static content it serves.

You can also get a lot of mileage just out of good use of solid color fills in your web page; these are specified in HTML and don't require serving up any additional content. Many embedded web devices with simple processor cores do their user interface mostly with color fills, and small bits of graphics in .gif format to improve the look. Consider this web page:


Basic Setup - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Google
LINKSYS
A Division of Cisco Systems, Inc.
Firmware Version: v1.00.6
Wireless-G Broadband Router
WRT54G
Setup
Setup
Wireless
Security
Access Restrictions
Applications & Gaming
Administration
Status
Basic Setup | DDNS | MAC Address Clone | Advanced Routing
Internet Setup
Network Setup
Done

http://192.168.11.1/basic.htm

Internet Connection Type	Automatic Configuration - DHCP
Optional Settings (required by some ISPs)	
Router Name:	WRT54G
Host Name:	
Domain Name:	
MTU:	Auto
Size:	1500
Router IP	
Local IP Address:	192 . 168 . 11 . 1
Subnet Mask:	255 . 255 . 255 . 0
Network Address Server Settings (DHCP)	
DHCP Server:	[✓ Enable] Disable
Starting IP Address:	192.168.11. 50

Automatic Configuration - DHCP: This setting is most commonly used by Cable operators.

Host Name: Enter the host name provided by your ISP.

Domain Name: Enter the domain name provided by your ISP.
More...

Local IP Address: This is the address of the router.

Subnet Mask: This is the subnet mask of the router.

DHCP Server: Allows the

This complex setup page includes two logo graphics (one is scrolled out of view) and various navigation controls, yet only involved the use of nine small .gif files *totaling* 9,755 bytes.

CSS: getting control of your layout

When I was talking about HTML, I mentioned that its original purpose was a markup language for simple text documents with cross-reference links. This quickly became insufficient for the people who wanted the web to be more visual and graphical, because HTML wasn't designed for that. There were various tricks you could use to kind of place things, but they involved heavy use of tables and lots of oddly split graphic files and were fairly hard to do well. Recent web browsers, however, give the web page much more control, through a technology called CSS.

CSS stands for "Cascading Style Sheets," which doesn't sound like anything helpful at all. But it turns out to be a powerful set of extensions to HTML that let you define the visual style of, well, just about anything, in terms of its size and font and color and spacing and padding and background and borders and lots of useful things like that. It lets HTML stay focused on its original intent: text with markup, that "falls back" gracefully from anything you ask it to do that it doesn't understand. But CSS provides essentially a "side language" just for describing how things should look, and you can both use these descriptions within tags and set up rules for how the browser should react to tags when it encounters them.

This sounds pretty abstract, so let's give some examples. Here is the HTML for a paragraph with some style instructions:

```
<p style="font-family: Verdana, Arial, Helvetica, sans-serif;
font-size: 12pt; color: #C04040;">This is text with style!</p>
```

This simply says that this paragraph tag, in addition to any formatting rules it already has, should use the Verdana font (or, if the computer doesn't have Verdana, then Arial, and so on down the line), should use a twelve-point font size, and should be a particular color (a certain shade of red, in this case). These styles apply to this paragraph and only this paragraph; once the tag is closed (with </p>) the style no longer applies to anything.

If you like to use that particular paragraph style, you can define a class for the style that you can use in several places. Each use simply refers the browser back to your definition of the class. You use a class like this:

```
<p class="stylish">This is text with style!</p>
```

Earlier in the HTML (usually in the <head> section) you would have defined the style, in code such as this:

```
<style><!--
p.stylish { font-family: Verdana, Arial, Helvetica, sans-serif;
font-size: 12pt; color: #C04040; }
//-->
</style>
```

Or, if you wanted to use that style in many different pages, you could put it and other style rules into their own file, called a "style sheet," which you simply tell the web browser to load as part of its rules. A single line in the <head> section tells the browser to load the rules:

```
<link href="mystyles.css" rel="stylesheet" type="text/css" />
```

These various style instructions follow rules of precedence. Any style attributes in a tag extend or override the rules for a tag within the file or style sheet, which extend or override any rules for the same class or tag encountered earlier in a file or style sheet, which extend or override the rules that your web browser has for defaults. (And there are more rules than that.) The idea is to let you set up general rules for a site, more focused rules for a page, and tightly focused rules for one single element (pretty much any tag) on the page, as needed.

There is a long list of style settings to control not just colors and fonts but sizes and placements of elements, and graphics to use for backgrounds, and lines and borders and fills. Most modern business web sites make good use of CSS for their layout, and your user interface can certainly benefit from these approaches as well. Taking some time to learn CSS gives you more control of your visual layout than was possible in HTML alone.

A recent book on HTML should discuss CSS and how to use it. If you need to go into greater depth, consider a book specific to CSS itself (see my references at the end), or search the web for CSS resources.

JavaScript: a programming model for the browser

Modern web browsers typically support JavaScript, an entire programming language (or scripting language, if you prefer) that runs inside the browser on behalf of web pages.

The JavaScript language draws much of its syntax from Java or C++, so it's not hard for a programmer familiar with one of those languages to quickly pick up the basics. JavaScript is an interpreted language (deciding what to do as it runs rather than compiled ahead of time) and its variables are quite loosely typed (like the "variant" type in Visual Basic). The JavaScript language itself isn't anything special; rather, the value of JavaScript lies in its ability to directly examine and manipulate the display elements and react to browser events.

I'm not even going to try to present an overview of JavaScript in this paper, because there's a lot to it. Rather, I simply want to mention two particular areas where it can be particularly helpful in a user interface.

For one thing, JavaScript is good for validating input. If you have a web form, some JavaScript code can sanity-check the user values before submitting them back to your device. If the code doesn't like the values, it can complain with message boxes or even changing the page display itself to call attention to the problems. This is usually a more responsive user experience than having your web server code generate entire error pages complaining about something it doesn't like on the previous screen.

Further, JavaScript can react to basic events like mouse clicks and drags and key presses, letting you, in essence, create your own "custom controls" for the page. (A word of warning here, though: some details of JavaScript events work quite differently in Internet Explorer than in pretty much any other browser, which necessitates some extra code to react properly across different browsers.) There is also an event for when the page has finished loading into the web browser, and you can set timers in JavaScript to cause further events later in time.

JavaScript and web page programming is a large topic, because to really use it you need to not only learn the language (not so hard) but also the browser's Document Object Model (DOM), the arrangement of objects that represent an HTML page on the browser display... and understanding DOM requires that you understand HTML and probably CSS first. I'll again recommend an O'Reilly book to help you tackle it, *JavaScript: The Definitive Guide* by David Flanagan.

(Microsoft offers a similar technology called VBScript: the same idea, with a Visual Basic style of syntax instead of a Java/C++ style. But I don't know any real advantage to VBScript on the web browser, and it only exists on Internet Explorer, so in general JavaScript is the better approach.)

Ajax: page updates when you want them

I didn't mention this specifically, but you may have already discerned this from my section on HTTP: web browsers only fetch pages from web servers when the user says so. The user either types a new URL in the address bar, or clicks on a link, and then the browser goes out to fetch and display the requested thing.

If you want your user interface to display information that is reasonably current, though, this may not be good enough. Even my simple light switch example may fall short: if some other user (or other input) changes the light switch, the display on the web browser is no longer correct. But how does the browser know differently? There is nothing in HTTP that lets the server push updates to the browser (and plenty of security reasons why it would be a bad idea if it could!). About the best you could do is use JavaScript to set a timer to periodically ask the browser to go refresh the page... which, by the way, is an annoying user experience if you're trying to do something when the page decides on its own to suddenly go load a new copy.

What is needed for a situation like this is some way for the web page itself to go and check with the web server on its own initiative, and if something is updated, to correct just that part of the display. Well, modern browsers offer this ability, and it's called Ajax.

Even though it isn't capitalized, Ajax stands for "Asynchronous JavaScript And XML." (I can see why the short name is better.) At its heart it refers to two key points: JavaScript is able to initiate a web fetch, reacting to the content when it arrives (without having to "hang" the browser waiting for the transaction to complete), and JavaScript is able to inject what it fetches into the page. (XML is HTML's younger and more ambitious cousin. Ajax references XML because one of the ways Ajax can update the display involves content that is in XML form rather than HTML, but in fact there is no requirement that you do this.) Ajax makes a request of a web server, using plain old HTTP (or HTTPS) like any other fetch, but your *code* rather than the browser gets whatever content is returned, to do with as it pleases. The content could be HTML to shove into the display, or plain text to replace a display element... or, the JavaScript code could parse what it gets back in any way that makes sense, to take whatever actions you like.

Since I've been pretty consistently recommending books for more information, it may surprise you that for Ajax I recommend that you *don't* get a book. There is likely enough information about Ajax in a good book on JavaScript; for example, chapter 20 of *JavaScript: The Definitive Guide*, fifth edition, contains all you would need. More to the point, a book on Ajax would probably present a lot of technologies built on top of Ajax that are unlikely to work with your particular web server.

So here's the basics: with only a minimum of cross-platform silliness, JavaScript code can use a special "XMLHttpRequest" object to do its own web fetch, and do whatever it likes with whatever the web server sends back. Since (again) *you* are writing the code for whatever dynamic content the web server sends back, that means you have complete control over both ends of the conversation. You can write tiny dynamic pages to give back just a snippet of text describing the current state or a snippet of HTML for rendering the current state. Returning to my light switch example, there could be a page called "lightstate" which returns as its content just a single word like this...

```
off
```

...and the JavaScript handler that receives this could simply shove this content within the <strong> tag that contains it, forcing the browser to update the display of just that one word.

Other technologies worth a look

Before we finish up, I simply want to list a few other topics you might care to investigate, although I won't go into any detail here.

One topic is *cookies*. A cookie is essentially a little bit of data that a web server asks the web browser to hold onto. The server includes this request as part of a reply header (so your code would need to be able to write these). The web browser subsequently includes any cookies that apply to a particular web server whenever it sends a request to that server, including them in the request header (so your code would need to be able to read these). This can be useful for tracking information over time, because otherwise there is no good way to tell users apart: every web request looks pretty much the same. Information on cookies can be found in a book on HTTP (discussing how they work within the HTTP headers) or in a book on JavaScript (since JavaScript on the web browser also has the ability to see, and set, cookies).

Another topic is *RSS feeds*. There is a growing interest in "publishing" changes to information using RSS (which stands for Really Simple Syndication), so that a user's RSS reader calls attention to changes the user cares about. This could be useful in a networked device. RSS really is "really simple;" it consists of plain old HTTP requests to URLs, except that these URLs are expected to give back a particular form of XML content that describes what information you have and when it was updated. There are some books on this subject, but also plenty of online descriptions if you search for them. Like I mentioned with Ajax, if you're writing the code that generates content in an embedded web server, it's no trick to also dynamically generate a page that works as an RSS feed.

You may also care to investigate more advanced client technologies: Java, Flash, Silverlight and so on. My advice is, *don't:* each one of these adds another significant skill set you'll need to acquire, and also reduces the number of customers who can use the pages you send. Now, there are plenty of people who would disagree with me on this position, particularly with regard to Flash, which has become quite widespread. If you listen to one of those people, I won't take it badly; as with any of the technologies I've thrown out in this paper, simply ask whether the advantages are worth the challenges, and plan accordingly.

Working with tight resources

Ever since the section on web graphics, I've been talking about things you can *add* to your web experience. But everything you add requires storage space and network bandwidth, and either or both of these may be at a premium for your device. So it's time to go on a diet: let's see what we can take away.

Where does your static content live? If it's on a hard drive then space may not be that tight... but if everything feeds from Flash memory, it's not hard to run out of room. (Worse still is if you make everything *just fit*... and then your first code update causes things to not fit any more!) If this is your situation, the biggest place to look carefully is at graphics, as they can take substantially more room than the text that makes up HTML and style sheets and JavaScript and such. (In such an environment, don't even think about including sound or video resources in your user interface!)

I'm not saying you need to eliminate graphics; I'm saying you need to economize. That corporate logo could probably be smaller, for starters. Use the right format for the job: photos should be in .jpg files, with the compression cranked up as high as you're willing to tolerate, while icons and simple graphics should be in .gif files, with the color palette shrunk to just what you need. If your graphics editor doesn't give you control over these details, you need to find a cheap one for web work that lets you economize rather than focus on beautiful image quality. That animated .gif file? Cute, but it would be smaller if it didn't move. Also, clever use of styles for background images can often let you get the same effect from one or two small graphics and possible a fill color as you might get from a much larger graphic. All of these save resource space, and most save network bandwidth as well. (The exception is that two small files may take more network traffic than one larger file, because of the HTTP headers and additional back-and-forth to set up the two requests.)

In terms of network bandwidth and/or page load speed, the best optimization is to recycle elements. If you can use the same graphic, or style sheet, or JavaScript file, in multiple pages, the web browser only needs to get it the first time, and can then reuse it from its cache.

In HTML files, look for places where you keep using the same "style" attribute settings, and make classes for these instead: it saves space, *and* makes the page more maintainable next year when you want to change the look. If you're using the same class definitions or other style rules in lots of pages, pull them out into a common style sheet. Similarly, if you keep using the same JavaScript functions, give them their own file.

If space is still tight, look at white space in HTML, CSS and JavaScript. If you're following book examples you may well have lots of spaces and line breaks and indents to make everything neat and readable and maintainable. But that's resource space and bandwidth that doesn't do a thing for your user. My suggestion: maintain your static resource files with lots of white space for your readability and maintainability, but have a utility to strip all of that out for the versions that ship out in your product. Do a web search for "remove HTML white space" for tips and tools.

Or go tighter still. To save storage space (although it does nothing for bandwidth), consider storing all of your static *text* content in .zip file form. HTML, JavaScript and CSS all compress nicely into .zip files, and you may find that the unzip code (which can be found online) takes up less Flash memory than the uncompressed content does. It's extra work for your processor to do, but in many cases it's worth it. But don't bother zipping .gif or .jpg files; they don't usually benefit much if at all.

And, of course, look at your code for generating dynamic content. You may find that the same snippets of HTML appear a lot in your code; some careful functionizing might help, or just putting heavily used bits of HTML into a table of strings that you can access, reducing the space for redundant string constants.

These are my suggestions; the documentation for your web browser may offer other suggestions. Any embedded design requires some attention to resource limits, and a web user interface is no different.

Conclusion (or: Wow, where do we start?)

I set out to write a paper, not a book, so I have necessarily only scratched the surface of a lot of different topics related to using a web browser as the user interface for an embedded web device. Unless much of this material is already familiar to you, you may be reaching the conclusion that this approach is really quite involved and complicated, and therefore expensive to develop. But that's not the conclusion I want to leave you with. Sure, there *can* be a lot of complexity if you are using all of these pieces. But that isn't where you start. Remember my light switch example: this stuff *can* be quite basic.

Two pieces of knowledge are essential. You've got to understand HTML well enough to hand-code a web page, and you've got to understand the programming model(s) specific to the web server available for your embedded operating system or network code. The first of these skills can be picked up from any number of books or even online tutorials; don't let my fondness for the "Definitive Guide" books from O'Reilly dissuade you from any other book that can quickly teach you HTML coding. The second of these skills involves the sometimes painful process of making sense of what your network code vendor has given you. Obviously if your web server solution is not that widely used, you may have trouble finding answers to your questions; I hope that my discussion of HTTP will give you some points of reference in figuring out what you've been given.

As for the rest... don't sweat it until you know you need it. And even then, sometimes you can get the answers you need faster by looking at what someone else has done than by reading a book. Essential tip: you can look at other people's HTML code for the pages you visit on the web. Every web browser I know of on PC, Mac or Linux has a menu command or a context menu choice to view the source code for a web page. That gets you the HTML code. If the code refers to an outside CSS style sheet or a JavaScript file, just type its URL into your browser and you can see that file too. (Remember, if the URL doesn't start with a computer name, then you need to attach the path or file you are given to what you already have, the same way the browser does when evaluating incomplete links.) These tricks let you see everything the web browser sees to render the page: forms and JavaScript validation, styles and graphics, you name it. It can be overwhelming, because many good web sites use a *lot* of style rules and JavaScript code snippets... but, like the accidental tangents that happen when searching the web, you can also stumble across a lot of good ideas on the way to the answer you need.

One other resource I might as well mention, since I've taken you this far... if you have a question, ask me. My e-mail address is mark@markbereit.com. I may not get to your question right away, but if I can offer advice or help I will.

There may not be many resources for your particular web server module, but there are plenty of resources for the rest of the web technology, because it's not specific to embedded products and so lots of people are doing it. And, conveniently enough, every one of the technologies I've talked about starts out at a level that is pretty simple and understandable, letting you "get your feet wet" while finding out how far you really need to take it.

So, the bottom line: most web technology is built up of a lot of little pieces that can be learned, and played with, with very little investment of time or money. These pieces can be leveraged through embedded code, as part of a web server component, to provide a user interface for an embedded device that may supplement, or replace, physical controls and displays. This embedded code tends to be quite straightforward as well, since it is essentially simple character output in reaction to the device state and/or simple character inputs received as part of the request. Networked embedded devices are increasingly taking advantage of the relative simplicity of development, modest resource requirements and (in some cases) cost savings to improve the products... and you can do this too.

Good luck!

References

Musciano, Chuck, and Bill Kennedy. *HTML & XHTML: The Definitive Guide*. 6th ed. O'Reilly Media, 2006.

Gourley, David, and Brian Totty. *HTTP: The Definitive Guide*. O'Reilly Media, 2002.

Meyer, Eric A. *Cascading Style Sheets: The Definitive Guide*. 3rd ed. O'Reilly Media, 2006.

Flanagan, David. *JavaScript: The Definitive Guide*. 5th ed. O'Reilly Media, 2006.

Bentham, Jeremy. *TCP/IP Lean*. 2nd ed. CMP Books, 2002.