Web Enabling Your Products Without Writing Any TCP/IP Code

AUTHOR
Jack Schoof, Chairman, CEO
NetMedia, Inc.
10940 N. Stallard Place
Tucson, AZ 85737
520-544-4567/0800 Fax
jschoof@netmedia.com
www.netmedia.com

ABSTRACT

This tutorial presents ideas and processes for web enabling products with the least amount of work expended on your part. The web server coprocessor is introduced along with schemes for enhancing web pages for use in embedded environments. Some of the information disclosed may be the subject of a pending patent.

AUTHOR BACKGROUND

Founder of Artisoft, Inc. and author of LANtastic, the most popular peer to peer network operating system in the world with over 10 million copies installed worldwide. Jack began Artisoft with $40,000 and within a few years exploded into 660 employees in 18 countries doing over 85 Million dollars per year. At one point Artisoft was the third largest Ethernet card supplier in the world. Artisoft created the first Ethernet adapter on a chip and was the first company to commercially put voice on a data network. Artisoft was the #1 company in America in 1992 (Business Week Magazine), and a record five times in a row on the Inc 500/Inc. 100 list of top companies. Mr. Schoof was named Entrepreneur of the Year for the Southwest, CRN's top 25 computer executives, and received PC Magazine's Technical Excellence Award for its voice over LAN product.

WELCOME

I guess I come qualified to talk about explosive markets, and networking. Artisoft was a wild ride. My focus has always been to bring technology to the masses. LANtastic was an outgrowth of my desire to make networking simple and more affordable for computer users. Nowadays, the Internet has taken over as the new networking bonanza, since peer to peer networking is everywhere today.

We are gathered here at this conference with the goal to embed the Internet inside everything from Boom Boxes to X-ray machines as cheaply as we can and as fast as we can. So what is the problem? 1) We've got protocols designed for mainframes, 2) Application Programming Interfaces (API's), from Berkley right after the 60's, 3) We've got multitasking issues and reentrancy, blocked and unblocked, Sockets and Winsock, Linux and Windows and Mac – Oh my. And to top it off we have Al Gore claiming he invented the internet.

EXISTING INTERNET PROTOCOLS

We should briefly look at the protocols for a minute and see what they are all about. First we have ARP which is Address Resolution Protocol. Mainly for Ethernet users, this means that the TCP/IP address needs to be converted to a real Ethernet address on the network so you can talk to an Ethernet card. Next there is Internet Control Message Protocol (ICMP), it can do many things but mainly you use it when you Ping a computer or a site to see if it is there. Next you have IP which is the Internet Protocol. It is a wrapper for other protocols providing the addressing scheme of the famous four bytes separated by dots. UDP is User Datagram Protocol. It is a simple protocol that just sends packets unacknowledged from one IP address to another. UDP is mainly used for its speed in low error rate environments like initial installation and configuration of embedded devices. TCP is the Transmission Control Protocol and is the main focus of the embedded designer. An important and often overlooked protocol for the embedded designer, is BOOTP or Dynamic Host Configuration Protocol (DHCP), which is a way to get your IP address from another computer without a factory set IP address. Last but not least there is the HyperText Transfer Protocol, (HTTP) which is how browsers request hypertext (HTML) and graphics from servers using Uniform Resource Locators (URLs).

If you know the alphabet soup intimately that is great, if you didn't, even better, because this tutorial hopes to show there are now methods available that you do not have to write any TCP/IP code in order to have a web enabled product. When I say TCP/IP code, I mean either underlying operating system type stuff where you are banging packets around yourself, or an even higher level such as the socket layer.

TRADITIONAL WEB ENABLING METHODS

OK, so how do we embed a web server? Some traditional ways are to find yourself a Real Time Operating System (RTOS) that works on your processor of choice. License the thing and then go to school to learn how to make it do all the neat things that you want it to do. An RTOS needs code space and ram space for itself, so unless you have code space and ram space free (if you do, the marketing people were not quite finished with the features, now were they?), your design hardware will have to grow. You might even have to chuck your happy cheap little processor to get a big one just to run the RTOS. You will also have to be a slave to the RTOS and learn its care and feeding. You have to free up time from your formerly dedicated application to give the RTOS time to process Internet packets, at the same time providing snappy response for your actual device. Many RTOSes bog down dramatically with high Internet traffic, so don't be surprised if you have little left for your actual application. If your keypad doesn't read correctly or the update of the display is not as speedy as before, these are some of the symptoms. Plus an RTOS typically runs a Sockets API interface which means that you still have to know all about TCP/IP to speak over the internet and that is not for this discussion, so we will RTOS this one out.

You could just purchase a controller with Internet protocols already embedded in it. This is similar to the RTOS concept in a chip, but you may still need to write Sockets code, or a server, or merge your code into another code environment. Still requires some TCP/IP code.

A way to embed the internet in your device without writing TCP/IP code is to cheat a little and embed just a little bit of code inside your processor and communicate with a gateway of some sort that does all the dirty work. The gateway could be a PC running Windows or Linux or some specialized hardware that performs the Internet protocols. Now you have at least two projects – the mini network that talks to the gateway, and the Internet network. This scheme could also be an installation nightmare for your customers. You mean my toaster requires me to run Linux? If this gateway is external to your device, you need some way to communicate with it. The Gateway now needs some setup and a user's manual separate from your device.

INTRODUCING THE WEB SERVER COPROCESSOR

The other way to do this is to combine all the good features of all these possibilities into a single package. I call it a web server coprocessor. The job of this coprocessor is to do all the dirty work of all the protocols in a package so tightly woven it is a complete server in and of itself.

The job then of your processor is just to send data values to the coprocessor when something changes like a temperature or status.

The web server coprocessor contains all the protocol software and interface hardware necessary to connect to the Internet. This could be either Ethernet or a serial connection for dial up situations. It provides the complete Internet server implementation including web page storage. It is a complete system with no support required from any other device, other than routing hardware or modems.

You can build or buy a web server coprocessor. If you build one yourself, you will need to go through the pain and suffering of the RTOS and other issues outlined above. The good news is that once you do it then you can use it again and again for all your other projects easily. Of course if you buy one, then you don't have any suffering and can concentrate on your device.

The web server coprocessor approach also allows you to make the web an option. If the customer buys the web enabled version, then you simply plug in the coprocessor.

DEFINITIONS

For the purpose of this discussion, I will refer to the device processor as the processor that you use to perform the device's main function. This could be an industrial controller, printer, dishwasher, test equipment, you name it. The coprocessor will be the web server processor used for web communication.

DATA OBJECTS FOR COMMUNICATION

So how do data values get from the device processor to the coprocessor? A standard serial port is the easiest. Then we need to have some way that the device processor and the coprocessor know what each other is talking about. This will be through data objects with data types like bytes, integers, long integers and strings. Don't worry, doing this in a processor is very simple. Typical object transmission and reception code for a device processor implemented in an 8051 is only 128 bytes of code or less.

What I am referring to by objects is a group of data that has a length and address which are known by both processors. The objects are created or used by the device processor, and stored for display or retrieval by the web server coprocessor. Think of these objects as a memory array or mail boxes through which the device processor and coprocessor communicate.

$$\begin{tikzcd}
 & \text{Object1} & & \\
\text{Device Processor} \arrow[r, leftrightarrow] & \text{Object2} \arrow[r, leftrightarrow] & \text{Web Server Coprocessor} \arrow[r, leftrightarrow] & \text{Internet} \\
 & \text{Object3} & &
\end{tikzcd}$$

Each processor can read and write into this object memory. The device processor places data from the application into these objects, and the server coprocessor takes the binary data converts it to human readable form and sends it to browsers when they request it. Conversely, when a user at a browser sends some data into the server via a form or other method, the data from the form gets moved into the object memory for the device processor to read.

To move data in and out of this object memory from the device processor we need just 3 commands, Write Object, Read Object and Status. Since there are only 3 commands, we can use the rest of the command byte to be the number of bytes in the message. Status is used for times when the web server coprocessor is busy doing something, or to let you know when new objects have come in from the Internet.

REAL WORLD EXAMPLE

So let's say that we are making a web enabled thermostat. It will take the temperature and broadcast it to the world. We will need a few objects for this thermostat, so lets define them.

Name	Size	Location	Initial Value
CurrentTemperature	Byte	0000	N/A
SetPoint	Byte	0001	72
Fan	Byte	0002	0
Unit	Byte	0003	0

All these objects are bytes since we do not expect temperatures in the house less than 40 degrees and greater than 100. To send one of these objects to the coprocessor we just need a four byte command: Write byte object, the address, and the byte itself. At 19,200 baud this can occur 480 times per second, and at 115,200 baud it can occur at almost 3000 times per second. This is plenty fast for most data updates.

$$\begin{tikzcd}
\begin{array}{l}\text{Thermostat}\\ \text{Processor}\end{array} \arrow[r, leftrightarrow] & \begin{array}{l}\text{CurrentTemperature}\\ \text{SetPoint}\\ \text{Fan}\\ \text{Unit}\end{array} \arrow[r, leftrightarrow] & \begin{array}{l}\text{Web Server}\\ \text{Coprocessor}\end{array} \arrow[r, leftrightarrow] & \text{Internet}
\end{tikzcd}$$

A simple thermostat algorithm (assuming air conditioning) looks like this:

```
Do
  Call Get_Temperature(CurrentTemperature)        'get the current temperature somehow
  Call Write_Object(CurrentTemperature)           'send over the temperature to the
coprocessor
  Call Read_Object(SetPoint)                  'get the SetPoint from the coprocessor
  If CurrentTemperature > SetPoint then            'above the setpoint?
      Call On(Unit)                                'then turn on the unit
      Call Write_Object(Unit)                 'tell the coprocessor
      Call On(Fan)                                 'and the fan
      Call Write_Object(Fan)                  'tell the coprocessor
  ElseIf CurrentTemperature < SetPoint then        'colder than the setpoint?
      Call Off(Unit)                          'yes, then turn off the unit
      Call Write_Object(Unit)                 'tell the coprocessor
      Call Off(Fan)                                'and turn off the fan
      Call Write_Object(Fan)                  'tell the coprocessor
  End If
  Call Sleep(60)                                   'sleep for a minute
Loop                                          'do this forever
```

We make a decision about temperature every minute and send the values to the coprocessor about current temperature and unit and fan state. As you can see the thermostat processor is asleep most of the time. Yet the web server coprocessor can be sending the pages to many viewers while the device sleeps.

<u>EMBEDDED HTML EXAMPLES</u>
Now how does the web server coprocessor know to display the temperature in the correct spot in the web page? The web server coprocessor needs to have some object output routines as it is emitting the page. And a way to know that it needs to output an object. If the following string was in an HTML file on the web server coprocessor:

The temperature is **^CurrentTemperature**

The coprocessor will see the up-arrow, and take the next word to see if it is an object name. If it is, it will go to the object's storage location in the coprocessor's memory, get the latest copy of the temperature, and emit it as the text in the sentence. So the HTML that gets emitted is:

The temperature is 78

Notice there was no java, no CGI, no funny business. A pointer in the HTML text links directly to an object name. Now anywhere that the up-arrow temperature is used, the characters for the current temperature will be emitted by the server. Users can ask for the page containing the temperature reference hundreds of times in a minute, but the device processor has not been bothered for any of the requests.

<u>GRAPHICAL EXAMPLE OF OBJECTS</u>
For a graphical example, assume we have a switch called motor. The value of motor will be 1 for on and 0 for off. A filename of a graphic can be called up by using:

<img src="/images/switch**^motor**.gif">

The **^motor** will be converted to a one or a zero depending on the value of the object. Then all you need to do is provide a graphic called switch1.gif and switch0.gif on the server. The switch1.gif can be a running animated motor and the switch0.gif can be a static motor.

So let's take a more complicated example. Say you want a graphical knob which can rotate into eight positions. You would simply build eight graphical pictures of the knob with the pointer in each of the positions. Then the data variable that you use for the position of the knob would be tacked on the end of the file as before:

Src="/images/knob**^position**.gif"

The object called position would be converted into the number 1 to 8 and the corresponding graphic file knob1.gif through knob8.gif would be displayed. Again all the device processor has to do is send the position object to the web server coprocessor whenever it changes.

HTML has radio buttons and check boxes. For these to be emitted by an embedded system, the object must return a text string "checked" when the box should be checked.

<input type="checkbox" name="fan" value="1" **^fan**>

Since the object system knows that the fan object is a checked object, then when the fan byte is non zero, **^fan** will return the string CHECKED and when fan is zero then the string returned will be null.

Objects can be more complicated. They can be numbers, strings, specific digits within numbers, they could be even I/O ports or internal web server parameters like IP address, time of day, or page hit counters.

<u>DATA INPUT FROM USERS</u>
Data output is important, but there are plenty of times you need input from users. This could be data input from a form, or the click of a button. Forms allow a browser to package up text and numbers and list selections, radio buttons and check boxes and send them to a web server.

Back to the thermostat, the device processor's job is to make the room temperature match the set point that came in from the web. The HTML for a text input field for the set point in a web page looks like:

```
Set Point   <input type="text" name="SetPoint"><br>
Fan
  <input type="radio" name="fan" value="0">  off
  <input type="radio" name="fan" value="1">  on
```

When the user enters a setpoint of 75, and clicks the fan on and presses submit, a packet is sent to the web server coprocessor with the following text embedded in it:

SetPoint=75?fan=1

The web server coprocessor cracks this string finding all the object names that it knows about and places the data in the objects. The next time the device processor goes to read the setpoint and the fan, it will have the new numbers. Reading objects by the device processor use basically the same command structure as sending objects. A command, address and data size is requested from the server coprocessor, and the bytes of the object are returned.

TASKS FOR DEVICE PROCESSOR

To review, what pieces does a device processor need to do to get on the web? It only needs to have an object transceiver module that can send and receive objects. The device flow is:

```
Do
  Perform device functions
  Check for Input
  Send changed Objects
Loop
```

If the object system is interrupt driven, then the object processing can be done in the background with very little processing power required.

SUMMARY

The goal of the web server coprocessor is to make communications with a device as easy as possible. This provides maximum capabilities with minimal learning and effort. With some special HTML techniques in the coprocessor, displaying objects and manipulating graphics is also easy. This example showed no Java or special functions within the web pages, yet complicated real-time graphics and displays can be built. A web server coprocessor does not preclude the use of Java or other means.

I believe that the web server coprocessor concept can speed your development of web enabled devices. Both from a standpoint of physical hardware, but also because the software and web page work is much easier. Even existing products with a serial port can now become web enabled easily.