The EtherSmart/WiFi Wildcard User Guide

V2.0 April 2007

Chapter 1

The EtherSmart/WiFi Wildcard User Guide

This document describes how to use the EtherSmart and WiFi Wildcards. The EtherSmart Wildcard provides wired Ethernet connectivity, and the WiFi Wildcard provides wireless 802.11b/g connectivity. These two distinct Wildcards are implemented using similar hardware, and a common software driver supports both.

The EtherSmart Wildcard™ implements a hardware/software Ethernet interface that provides communications between your instrument and other computers or peripherals via a Local Area Network (LAN) using the standard packet-based Ethernet signaling protocol. Simply plug an Ethernet cable into the Wildcard via the standard RJ-45 Ethernet connection.

The WiFi Wildcard™ implements a wireless interface that enables communications between your instrument and other computers or wireless access points via a Wireless Local Area Network (WLAN) using the standard 802.11b/g packet-based protocol.

Built-in software lets you send emails from the instrument, serve out static and dynamic web pages to your PC-based browser, and implement serial data exchanges with peripheral devices (this is known as "Serial Tunneling"). These tiny 2" by 2.5" boards are members of the Wildcard™ series that connects to Mosaic controllers.

This document describes the capabilities of the EtherSmart and WiFi Wildcards, tells how to configure their hardware, presents a description of the unified driver software, describes a demonstration program, and includes hardware schematics. A separate EtherSmart/WiFi Glossary provides a detailed description of each device driver function

Ethernet and WiFi

The EtherSmart Wildcard™ and the WiFi Wildcard™ let you put your instrument "online" so it can communicate with other computers on a Local Area Network (LAN) or Wireless Local Area Network (WLAN). The 2" by 2.5" EtherSmart and WiFi products are members of the Wildcard family that connects to Mosaic controllers. You can plug one or more EtherSmart or WiFi Wildcards into any QCard, QScreen, QVGA Board, PDQ Board, PDQScreen, or Mosaic Handheld system to deliver the benefits of Ethernet or WiFi connectivity to your application.

With the EtherSmart or WiFi Wildcard, your application program can send emails to alert other computers on the network when significant events occur. You can "browse into" your instrument using a web browser running on an online PC to monitor the status of your instrument. The instrument can implement "Serial Tunneling" by initiating or accepting TCP/IP Ethernet connections and exchanging binary and/or ASCII text data with other devices on the local area network.

The EtherSmart Wildcard uses an Ethernet hardware implementation called the "XPort" made by Lantronix (www.lantronix.com). "Ethernet" is a packet-based computer networking technology for Local Area Networks (LANs). It defines wiring and signaling for the physical layer, and frame formats and protocols. The wired version of Ethernet used by the EtherSmart Wildcard is known as IEEE 802.3. The Lantronix XPort combines a processor, flash and RAM memory, Ethernet network interface controller, and RJ45 jack into a single component. The Lantronix device can support one Ethernet connection at a time. A UART (Universal Asynchronous Receiver/Transmitter) chip on the EtherSmart board provides the interface between the parallel Wildcard bus and the Lantronix device's serial data lines. Loss-less data integrity is ensured by the combination of built-in 64-byte data buffers in the UART plus hardware handshaking between the UART and the Lantronix device. The 3.3V supply that runs the Lantronix module can be shut down under program control while the interface is not in use to conserve power.

WiFi is short for "wireless fidelity"; some readers will recognize this as a play on the "HiFi" nickname that was originally given to "high fidelity" stereo audio systems. WiFi is a wireless communications standard that allows the same type of packets sent via Ethernet to be sent wirelessly. WiFi is a packet-based computer networking technology for Wireless Local Area Networks (WLANs). It defines signaling for the physical layer, and frame formats and protocols. The protocol used by the WiFi Wildcard is known as IEEE 802.11b/g.

The WiFi Wildcard uses a hardware implementation called the "WiPort" made by Lantronix. The XPort and the WiPort are controlled in a very similar way by the Mosaic controller, and this enables a common set of driver functions to control either the EtherSmart or WiFi Wildcards.

Most of the features described in this document apply to both the EtherSmart and WiFi Wildcards. A single software driver supports both of these Wildcards. Because the EtherSmart Wildcard was produced first and the names were left intact, the majority of these driver functions start with the "**`Ether_`**" prefix. For example, the **`Ether_Send_Email`** function can be used to send an email via the EtherSmart Wildcard or the WiFi Wildcard.

In this document, we use the expression "EtherSmart/WiFi Wildcard" when no distinction needs to be made between the two types of Wildcards.

Software driver functions that start with "**`WiFi_`**" apply only to the WiFi Wildcard, and cannot be used with the EtherSmart Wildcard. For example, the **`WiFi_Security`** function configures the wireless security options for the WiFi Wildcard; these wireless options are not implemented on the EtherSmart Wildcard.

Multiple EtherSmart and WiFi Wildcards can be installed on a Mosaic controller. The EtherSmart and WiFi Wildcards are distinguished from one another when each is initialized. A parallel set of initialization routines is provided, some starting with "**`Ether_`**" to initialize the EtherSmart Wildcard, and others starting with "**`WiFi_`**" to initialize the WiFi Wildcard. The initialization process marks the identity of the Wildcard. For example, the **`Ether_Setup`** function initializes the

EtherSmart Wildcard using specified memory parameters, while the parallel function **WiFi_Setup** initializes the WiFi Wildcard.

Terminology Overview

Accessing the Internet and the World Wide Web ("the Web" for short) can quickly lead to an alphabet soup of protocol names. This section introduces some relevant terminology.

A LAN (Local Area Network) is a group of interconnected computers with a "gateway" computer that typically serves as a "router" to direct traffic on the LAN and between the LAN and other networks. Each computer on the LAN must have a unique 32-bit "IP address" (Internet Protocol address). An IP address is typically written as 4 decimal numbers (each between 0 and 255) separated by periods. For example:

10.0.1.22

The EtherSmart/WiFi Wildcard can be assigned an IP address explicitly by calling some configuration functions, or it can get its IP address automatically via "DHCP" (Dynamic Host Configuration Protocol) running on the gateway computer. The assigned IP address can be associated with a computer name (so you don't have to type numbers in your browser's address bar if you don't want to). The name can be assigned by asking your LAN system administrator to make an entry in the DNS (Domain Name Service) config file. Alternatively, you can create a local name on your PC by editing the hosts file; on a Windows XP machine, this file is found at:

C:\Windows\system32\drivers\etc\hosts

The format of the communications among the computers on the LAN is defined by various "protocols". The fundamental point-to-point connection protocol is called "TCP/IP" (Transmission Control Protocol/Internet Protocol).

"Serial tunneling" is a name for a simple exchange of serial data between two computers, typically using the raw TCP/IP protocol. Other protocols build on TCP/IP. For example, World Wide Web traffic uses "HTTP" (Hyper Text Transfer Protocol). Email uses "SMTP" (Simple Mail Transfer Protocol).

Web pages that are served out using HTTP are typically formatted using the "HTML" (Hyper Text Markup Language) format. Many good books and online references describe HTTP and HTML. Most web pages are "static", meaning that their content does not change with each visit to the page. However, in the context of embedded computers, "dynamic" web pages that provide up-to-date status information about the computer's state (inputs, outputs, memory contents, etc.) are very useful. The driver code described in this glossary enables you to code both static and dynamic web pages.

The "embedded web server" that runs when you execute the EtherSmart/WiFi library code responds to information requests from your browser. You can create a set of web pages, each with a specified name, or "URL" (Universal Resource Locator) and an associated "handler function" that serves out the static or dynamic web content. A URL is a web page address as sent by the browser running on your desktop PC to the embedded web server. For the purposes of this document, the URL is the portion of the web address that is in the browser's address bar after the IP address or computer name. For example, if you have assigned IP address 10.0.1.22 to the EtherSmart/WiFi Wildcard, and you type into your browser's address bar:

```
10.0.1.22/index.html
```

then the URL as defined in this document is

```
/index.html
```

Each URL starts with a / character, and is case sensitive. Some URL's include a "query field" that starts with the ? character and contains fieldname and value fields resulting from "forms" that were filled out by the user in the browser window. The functions described in this glossary make it easy to extract data from these fields to direct the operation of the handlers. In fact, form data from the browser provides an excellent way for the web interface to give commands to the embedded computer (to take data samples, extract data from memory and report the results to the browser, etc.)

The web interface can be used to implement a "remote front panel" on instruments that contain a Graphical User Interface (GUI) and touchscreen. This feature allows a replica of the GUI screen to be presented in the browser window on a remote PC, and enables mouse clicks to mimic the action of touches on the instrument's touchscreen. A set of functions in this driver and a complementary set of functions in the GUI toolkit coordinate this capability. The relevant GUI Toolkit functions are presented in a separate glossary section at the end of this document.

WiFi Terminology

All WiFi nodes that share the same "SSID" (Service Set IDentifier) can associate with one another, and they can communicate if their security settings and security keys are compatible. The default SSID of the WiFi Wildcard is

```
WIFI_WILDCARD
```

SSID's are case sensitive.

A WLAN (Wireless Local Area Network) is typically operated in "infrastructure mode" meaning that a wireless "access point" (AP) coordinates messages among various WiFi nodes. A device that is not in infrastructure mode is in "ad hoc mode", meaning that it is configured for one-to-one communications with another WiFi device that is in ad hoc mode and that has the same SSID and security settings. WiFi devices communicate on channels within the 2.4GHz band. Channel numbers range from 1 to 13, with channels 1 to 11 allowed in the US and Canada. The channel numbers are typically automatically negotiated by the WiFi nodes, but the WiFi Wildcard allows you to specify the default channel when in ad hoc mode (the default is channel 11).

Because wireless signals can be intercepted remotely, securing the data traffic using encryption is wise. The original WiFi security suite is called "WEP", or Wired Equivalent Privacy. WEP is widely supported, but it is not a strong form of encryption and can be broken. The two key strengths of WEP are known as WEP64 and WEP128. WEP64 uses a 40-bit key concatenated with a 24-bit initialization vector, and WEP128 uses a 104-bit key concatenated with a 24-bit initialization vector.

An improved WiFi security suite is "WPA" (WiFi Protected Access). An even newer security suite associated with the 802.11i protocol is known as WPA2, and its associated encryption scheme is known as CCMP.

The encryption key for WEP and WPA can be entered as a hexadecimal value, or as a "passphrase". A passphrase is an 8 to 63 byte printable ASCII string that is processed by a "hash function" to create one or more numeric keys. If a passphrase is used, the WiFi Wildcard always uses key index 0 generated by the passphrase hash function.

Each security suite (such as WEP or WPA) can be configured in a number of ways. Configuration options include pairwise encryption method, group encryption method, and authentication using Pre-Shared Keys (PSK). See the "WiFi Wildcard Configuration" section for a description of these options.

Note that both the EtherSmart Wildcard and the WiFi Wildcard offer a completely independent means of AES (Advanced Encryption Standard) data encryption that is configured by the **`Ether_Encryption`** function; see its glossary entry for details.

The EtherSmart/WiFi Glossary Describes the Driver Functions

The companion to this document is the EtherSmart/WiFi Wildcard Glossary which describes each of the driver functions in detail. Browsing the "Categorized List of EtherSmart/WiFi Library Functions" in the Glossary will yield a good overview of the capabilities of the driver software.

Ethernet/WiFi Function Names

Except for initialization functions as noted in the glossary descriptions, functions that start with "**`Ether_`**", "**`HTTP_`**" or "**`HTTP_GUI`**" can be used to control both EtherSmart and WiFi Wildcards. Functions that start with "**`WiFi_`**" apply only to the WiFi Wildcard.

EtherSmart and WiFi Wildcards are distinguished at initialization time. For each of the EtherSmart initialization functions that start with the **`Ether_`** prefix, a parallel initialization function that starts with the **`WiFi_`** prefix is available. See Table 1-3 below to view a list of initialization functions for EtherSmart and WiFi Wildcards.

The WiFi Wildcard driver was coded as an extension of the driver for the EtherSmart Wildcard, and the original function names were left intact. Thus, it is best to interpret the prefix "Ether" as referring to the TCP/IP communications link which can be implemented via Ethernet or WiFi.

Internet Protocol Support

The EtherSmart and WiFi Wildcards implement the following protocols to establish and manage communications:

### TCP/IP (Transmission Control Protocol/Internet Protocol) enables reliable data exchange between the EtherSmart/WiFi Wildcard and a remote host with no lost data. Simple data exchanges with other Ethernet or WiFi devices use the basic TCP/IP protocol to establish a connection and transfer data. TCP/IP is the basis of data exchange protocols including Rlogin and Telnet which can be used to download code and interactively communicate with the operating system on Mosaic controllers via a simple network terminal such as Putty. This capability is explained in the section titled “Code Downloads and Interactive Communications via TCP/IP”. TCP/IP is also the foundation of higher level services including HTTP and SMTP.

### HTTP (HyperText Transfer Protocol) is implemented to deliver a dynamic webserver under the control of your application program. This webserver is available at the standard TCP/IP port 80 used by web browsers. You configure the webserver by defining and posting a “handler function” for each web address that is implemented. When the browser requests the web address (known as a URL, or Uniform Resource Locator), the EtherSmart/WiFi webserver automatically executes the handler which performs any required actions and transmits the web page to the browser. As described in a later section, versatile functions make it easy to use HTML “forms” to request particular data or stimulate particular actions from the Mosaic controller. An additional “configuration webserver” is built into the Lantronix device for low-level configuration of the Lantronix firmware, and is available at TCP/IP port 8000.

### SMTP (Simple Mail Transfer Protocol) is implemented to enable the EtherSmart/WiFi Wildcard to send outgoing email to a mail server on the LAN. The recipient, destination IP (Internet Protocol) address, subject and email contents can be dynamically controlled by the application program. Outgoing emails can be used by your instrument to send alerts or status updates.

### ICMP (Internet Control Message Protocol) allows the EtherSmart/WiFi Wildcard to send and return “ping” packets. These can be useful for network diagnostics. This low-level protocol is implemented by the Lantronix firmware.

### ARP (Address Resolution Protocol) establishes the correspondence between the 48-bit Ethernet Media Access Control (MAC) address which is visible on the Lantronix label, and the assignable 32-bit IP (Internet Protocol) address that identifies the parties on a network. This low-level protocol is implemented by the Lantronix firmware.

### DHCP (Dynamic Host Configuration Protocol) allows the gateway computer (or Wireless Access Point) on a Local Area Network to automatically assign an IP address to the EtherSmart/WiFi Wildcard as soon as it is plugged into the network. As described in a later section, an IP address can also be explicitly assigned using a web configuration screen, a telnet configuration session, or by calling an easy-to-use driver function from the application program. The low-level DHCP protocol is implemented by the Lantronix firmware.

Summary of EtherSmart/WiFi Capabilities

The following table summarizes the capabilities of the EtherSmart Wildcard.

Table 1-1 Capabilities of the EtherSmart/WiFi Wildcards.

Protocols	SMTP (outgoing email) , HTTP (webserver), TCP (serial tunneling), ICMP (ping), ARP, DHCP (automatic IP-address assignment)
Send Email	Delivers email with program-controlled content to specified LAN IP addresses
Exchange Data (Serial Tunneling)	Opens or accepts a TCP/IP connection to send and receive binary or ASCII data
Browsable	Accepts connections from your web browser, serving out static or dynamic web pages including HTML text, data, and images as specified by your program
Pre-coded Drivers	Pre-coded software manages device configuration, opens connections, exchanges data, sends email, parses web URLs, and serves out web pages

EtherSmart Wildcard Hardware

Figure 1-1 illustrates the hardware on the EtherSmart Wildcard. (The WiFi Wildcard is described in the following section.) The large component is the XPort from Lantronix, providing an Ethernet co-processor and network interface built into an RJ-45 connector housing. An Ethernet extension cable is available from Mosaic if you need to bring the female RJ-45 jack out to a panel connection on the front or back of your instrument.

A small linear regulator on the Wildcard converts the 5 volt power on the Wildcard bus to the 3.3V supply required by the XPort. The XPort draws about 250 mA of current. The regulator can be shut down under program control while the XPort is not in use to save 1.25 Watts of power.

The XPort exchanges data via a serial interface, while the Wildcard bus is a parallel interface. A UART (Universal Asynchronous Receiver/Transmitter) chip on the EtherSmart board implements the conversion between the parallel Wildcard bus and the XPort's serial data lines. The UART contains two 64-byte FIFO (First In/First Out) buffers, one for incoming data, and one for outgoing data. The XPort and the UART "handshake" with one another using "Ready to Send" (RTS) and "Clear to Send" (CTS) hardware lines. This scheme ensures that no data is lost due to buffer overflow in the UART or the XPort: the sending device stops transmitting until the receiving device indicates that there is room in its buffer for more data.

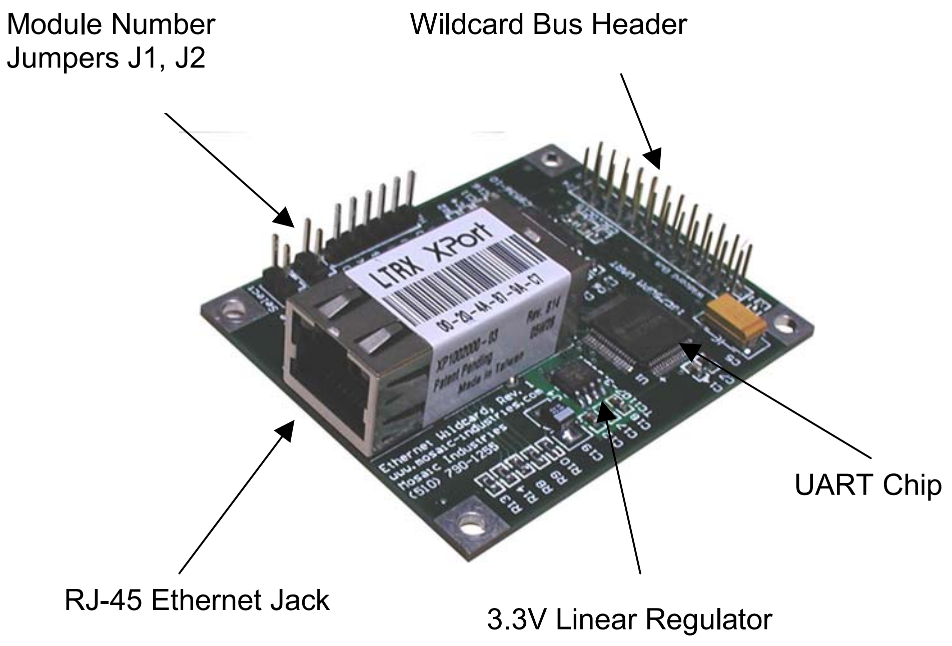


Figure 1-1 The EtherSmart Wildcard.

WiFi Wildcard Hardware

Figure 1-2 illustrates the hardware on the WiFi Wildcard. (The EtherSmart Wildcard is described in the prior section.) The large component is the WiPort from Lantronix, providing a WiFi co-processor and network interface built into a shielded housing with an antenna connector. A WiFi antenna is available from Mosaic; it can be mounted on the Wildcard in one of two ways. In Figure 1-2, the antenna connector from the WiPort is bolted through a mounting hole on the Wildcard, enabling the antenna to be screwed onto the board from the top. The antenna has a hinge that allows it to stand straight up or to be oriented parallel to the Wildcard. The alternative procedure is to mount a clamp (supplied by Mosaic with the WiFi antenna) onto the Wildcard, and use this to affix the antenna to the board. This allows more degrees of freedom in antenna orientation, and also eliminates the depth requirement below the Wildcard for the WiPort's antenna connection. For those who need a remote mounted antenna, a 2 foot extension cable is available from Mosaic.

The WiPort antenna connection terminates in a "Reverse Polarity SMA" (RPSMA) female connector. This is a bit confusing, as the "reverse polarity" designation means that a connector with a center pin (normally considered male) is called female. The WiFi antenna has an RPSMA male connector (no center pin). The optional extension cable has one RPSMA male end, and one RPSMA female end. The non-removable antenna cable on the WiPort module is about 4 inches in length.

A small linear regulator on the Wildcard converts the 5 volt power on the Wildcard bus to the 3.3V supply required by the WiPort. The WiPort draws about 300 mA of current. The regulator can be shut down under program control while the WiPort is not in use to save 1.5 Watts of power.

The WiPort exchanges data via a serial interface, while the Wildcard bus is a parallel interface. A UART (Universal Asynchronous Receiver/Transmitter) chip on the WiFi Wildcard board implements the conversion between the parallel Wildcard bus and the WiPort's serial data lines. The UART contains two 64-byte FIFO (First In/First Out) buffers, one for incoming data, and one for outgoing data. The WiPort and the UART "handshake" with one another using "Ready to Send" (RTS) and "Clear to Send" (CTS) hardware lines. This scheme ensures that no data is lost due to buffer overflow in the UART or the WiPort: the sending device stops transmitting until the receiving device indicates that there is room in its buffer for more data.

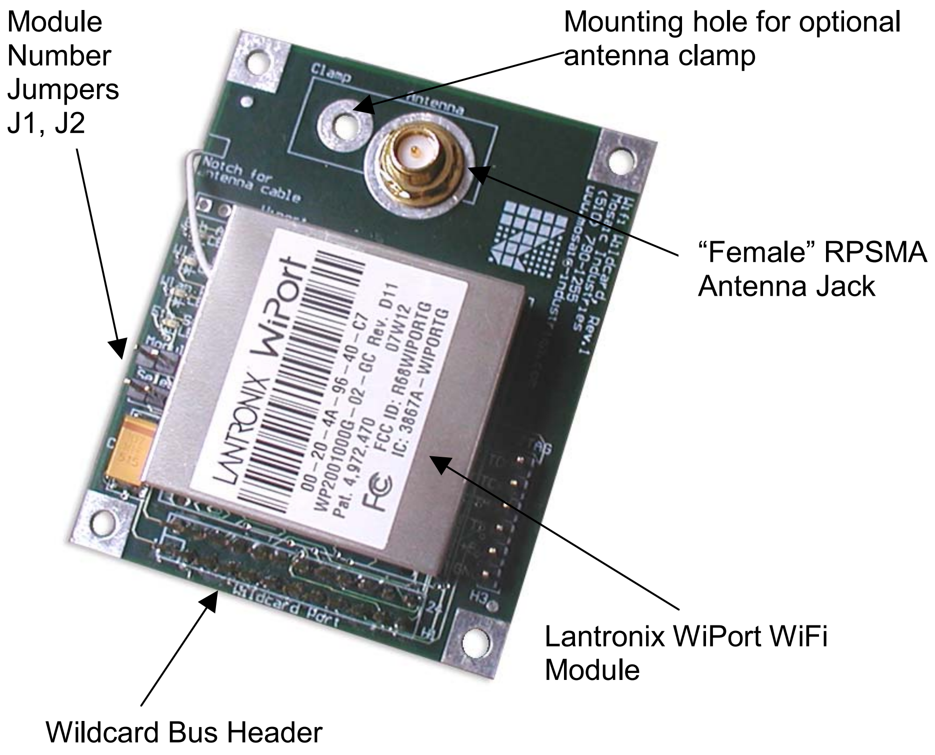

Module Number Jumpers J1, J2
Mounting hole for optional antenna clamp
"Female" RPSMA Antenna Jack
Lantronix WiPort WiFi Module
Wildcard Bus Header

Figure 1-2 The WiFi Wildcard.

Connecting To the Wildcard Bus

The 24-pin Wildcard bus header interfaces to the host processor (QCard, QScreen, QVGA Board, Handheld, or PDQ series controller). To connect the EtherSmart/WiFi Wildcard to the Wildcard bus on the controller board, follow these simple steps.

With the power off, connect the female 24-pin side of the stacking go-through Wildcard bus header on the bottom of the EtherSmart/WiFi Wildcard to Module Port 0 or Module Port 1 on the controller (or its mating PowerDock). The corner mounting holes on the Wildcard must line up with the standoffs on the mating board. The module ports are labeled on the silkscreen of the controller board. Note that the EtherSmart/WiFi Wildcard headers are configured to allow direct stacking onto the controller board, even if other Wildcards are also installed. Do not use ribbon cables to connect the Wildcard to the bus.

CAUTION: The Wildcard bus does not have keyed connectors. Be sure to insert the module so that all pins are connected. The Wildcard bus and the Wildcard can be permanently damaged if the connection is done incorrectly.

Selecting the Wildcard Address

Once you have connected the EtherSmart/WiFi Wildcard to the Wildcard bus, you must set the address of the module using jumper shunts across J1 and J2.

The Wildcard Select Jumpers, labeled J1 and J2, select a 2-bit code that sets a unique address on the module port of the controller board. Each module port on the controller accommodates up to 4 Wildcards. Wildcard Port 0 provides access to Wildcards 0-3 while Wildcard Port 1 provides access to Wildcards 4-7. Two Wildcards on the same port cannot have the same address (jumper settings). Table 1-1 shows the possible jumper settings and the corresponding addresses.

Table 1-2 Jumper Settings and Associated Addresses.

Wildcard Port	Wildcard Address	Installed Jumper Shunts
0	0	None
	1	J1
	2	J2
	3	J1 and J2
1	4	None
	5	J1
	6	J2
	7	J1 and J2

The Wildcard address is called the "module number" or simply "modulenum" in the input parameter lists of the software driver functions. Most of the driver functions require that the modulenum be passed as a parameter. Make sure that the jumper settings on the Wildcard hardware match the modulenum parameter that is specified in the application software.

Note that module number 0 is reserved on the QScreen and Mosaic Handheld controllers.

Terminology Overview

Accessing the Internet and the World Wide Web ("the Web" for short) can quickly lead to a proliferation of protocol names and acronyms. This section reviews some relevant terminology.

Ethernet, LANs, and IP Addresses

The Ethernet protocol defines a packet-based computer networking technology for Local Area Networks (LANs) that is broadly supported by desktop PCs and a variety of computer-based instruments. The Ethernet standard defines wiring and signaling for the physical layer, and frame formats and protocols. The wired version of Ethernet used by the EtherSmart Wildcard is known as IEEE 802.3.

A LAN is a group of interconnected computers with a "gateway" computer that serves as a "router" to direct traffic on the LAN and between the LAN and other networks. Each computer on the LAN must have a unique 32-bit "IP address" (Internet Protocol address). An IP address is typically written as 4 decimal numbers (each between 0 and 255) separated by periods. For example, the following is a valid IP address:

10.0.1.22

The IP addresses 0.0.0.0 and 255.255.255.255 are reserved and should never be assigned to a device on a LAN. The EtherSmart Wildcard can be assigned an IP address by calling some configuration functions as explained in a later section, or it can get its IP address automatically via "DHCP" (Dynamic Host Configuration Protocol) running on the gateway computer.

You must know your EtherSmart's IP address or its equivalent name before you can connect to it

EtherSmart system configuration and IP address assignment are discussed in more detail in the section titled "Assigning an IP Address to the EtherSmart or WiFi Wildcard".

The assigned IP address can be associated with a computer name (so you don't have to type numbers in your browser's address bar if you don't want to). The name can be assigned by asking your LAN system administrator to make an entry in the DNS (Domain Name Service) configuration file. Alternatively, you can create a local name on your PC by editing the hosts file. On a Windows XP machine, this file is found at:

C:\Windows\system32\drivers\etc\hosts

The "gateway" is the router computer on the LAN. The gateway routes to a subset of IP addresses specified by the zeros in the LAN's "netmask". The number of "subnet bits" is equal to the number of bits specified as zeros in the LAN's netmask. For example, setting the number of subnet bits to 8 is equivalent to specifying a "netmask" of 255.255.255.0, meaning that a maximum of 255 IP addresses can be represented on the LAN; this is a "type C" LAN. Specifying 16 subnet bits is equivalent to a netmask of 255.255.0.0, corresponding to a "type B" LAN. Specifying 24 subnet bits is equivalent to a netmask of 255.0.0.0, corresponding to a "type A" LAN.

The IP addresses on the LAN are typically accessible only to other computers on the same LAN. The gateway or router is the only computer on the LAN that is accessible to one or more networks in "the outside world". The gateway typically has one IP address for use on the LAN, and a different one that allow it to be accessed by other networks. Sometimes the external IP address of the gateway changes from session to session; this is how most "DSL" network connections function. To be able to connect to the EtherSmart or other computers on a specified LAN from an arbitrary computer on the Internet, you must know the externally available IP address of the LAN's gateway or router device. Talk to your system administrator about how to accomplish this if you need to access the EtherSmart/WiFi Wildcard from computers that are not part of the Wildcard's Local Area Network.

TCP and Serial Tunneling

The format of the communications among the computers on the LAN is defined by various "protocols". The fundamental point-to-point connection protocol is called "TCP/IP" (Transmission Control Protocol/Internet Protocol). TCP is a connection-based protocol that ensures that no data is lost as long as the connection remains open. If for some reason a packet sent by one computer does not arrive at the communicating partner, the TCP protocol re-transmits the packet until it is received intact.

When one host (computer, instrument, etc.) wants to communicate with a remote host via TCP, it requests a connection by sending a "SYN" (synchronization) packet to the remote's IP address. The remote accepts the connection by responding with a [SYN, ACK] acknowledge packet directed to the IP address of the requester, and the requester completes the process with an ACK acknowledge packet to establish the connection. This handshaking is performed by the Lantronix firmware and requires no intervention on the part of the programmer or the application. The application program simply uses a high level function such as **`Ether_Connect`** to open a connection as described in a later section, and then calls other high level functions to send and receive data.

For experts who want to view the raw TCP/IP packet exchange on the LAN, download the Ethereal network analyzer software that is available at www.ethereal.com. This free PC software can be very useful when debugging TCP/IP network application software.

"Serial tunneling" is a common name for the exchange of serial data between two computers, typically using the TCP/IP protocol. In network terminology, a "tunnel" is a path that passes data without the sender and receiver having to know or care about the details of the transfer mechanism. "Serial tunneling" was originally coined as a term to describe a method for allowing instruments and computers that were designed to communicate via RS-232 to now communicate via TCP/IP. Its meaning has broadened to encompass a variety of serial data exchanges between instruments via Ethernet and WiFi.

Outgoing Email

The SMTP (Simple Mail Transfer Protocol) email protocol builds on a TCP/IP connection. The EtherSmart/WiFi Wildcard can send outgoing email using SMTP. A single high level function named **`Ether_Send_Email`** handles all of the details for you. You pass to the function the gateway IP address on your LAN, and strings that specify the sender email address, recipient email address, and the email body. The Wildcard's driver software and the mail server running on the gateway computer cooperate to process the email. Sending email is explained in detail in a later section.

HTTP Web Service

World wide web traffic uses "HTTP" (HyperText Transfer Protocol) which builds on a TCP/IP connection. Web pages that are served out using HTTP are typically formatted using the "HTML" (HyperText Markup Language) format. Many good books and online references describe HTTP and HTML. Most web pages are "static", meaning that their content does not change with each visit to the page. However, in the context of embedded computers, "dynamic" web pages that provide up-to-date status information about the computer's state (inputs, outputs, memory contents, etc.) are very useful. The pre-coded driver functions enable you to serve both static and dynamic web pages.

By coding web content into your application, you can enable remote monitoring and control of your instrument from your web-connected PC. The "embedded web server" that runs when you execute the Wildcard's library code responds to information requests from your browser. You can create a set of web pages, each with a specified name, or "URL" (Uniform Resource Locator) and an associated "handler function" that serves out the static or dynamic web content.

A URL is a web page address as sent by the browser running on your desktop PC to the embedded web server. For the purposes of this document, the URL is the portion of the web address that is in the browser's address bar after the IP address or computer name. For example, if you have assigned IP address 10.0.1.22 to the EtherSmart/WiFi Wildcard, and you type into your browser's address bar:

10.0.1.22/index.html

then the URL as defined in this document is

/index.html

Each URL starts with a / character, and is case sensitive. Some URL's include a "query field" that starts with the ? character and contains fieldname and value fields resulting from "forms" that are filled out by the user in the browser window. The software driver functions make it easy to extract data from these fields to direct the operation of the handler functions. In fact, form data from the browser provides an excellent way for the web interface to give commands to the embedded computer (to take data samples, extract data from memory and report the results to the browser, etc.)

Web-Based Remote Front Panel

The web interface can be used to implement a “remote front panel” on instruments that contain a Graphical User Interface (GUI) and touchscreen such as the QVGA Controller or PDQScreen. This feature allows a pixel-by-pixel replica of the GUI screen to be presented in the browser window on a remote PC, and mouse clicks to mimic the action of touches on the instrument’s touchscreen. A set of functions in this driver and a complementary set of functions in the GUI toolkit coordinate this capability. This feature is described in a later section.

Function Naming Conventions

Functions that provide TCP/IP functionality, serial tunneling (point to point TCP/IP communications), and email start with the **`Ether_`** prefix; these functions can be used to control both EtherSmart and WiFi Wildcards.

As explained above, the EtherSmart and WiFi Wildcards share the same code base, but are distinguished at initialization time. The low level function **`WiFi_Module`** marks a specified module as a WiFi Wildcard. For each of the EtherSmart initialization functions that start with the **`Ether_`** prefix, a parallel initialization function that starts with the **`WiFi_`** prefix is available. The latter functions include a call to the **`WiFi_Module`** function. See Table 1-3 below for a list of initialization functions for EtherSmart and WiFi Wildcards, and consult the initialization function glossary entries for details.

Functions that implement world-wide-web functionality start with the **`HTTP_`** prefix. The HTTP protocol underlies the web data exchanges.

Functions that implement the web “remote front panel” feature for GUI-based instruments start with the **`HTTP_GUI`** prefix.

Ethernet/WiFi Function Names

Except for initialization functions (see Table 1-3 below), functions that start with "**`Ether_`**", "**`HTTP_`**" or "**`HTTP_GUI`**" can be used to control both EtherSmart and WiFi Wildcards. Functions that start with "**`WiFi_`**" apply only to the WiFi Wildcard.

EtherSmart and WiFi Wildcards are distinguished at initialization time. For each of the EtherSmart initialization functions that start with the **`Ether_`** prefix, a parallel initialization function that starts with the **`WiFi_`** prefix is available. Table 1-3 in the "Initialization, Configuration and Diagnostics" section below presents a list of initialization functions for EtherSmart and WiFi Wildcards.

The WiFi Wildcard driver was coded as an extension of the driver for the EtherSmart Wildcard, and the original function names were left intact. Thus, it is best to interpret the prefix "Ether" as referring to the TCP/IP communications link which can be implemented via Ethernet or WiFi.

Browser Notes

You'll use a web browser running on your PC to interact with the web server running on the EtherSmart/WiFi Wildcard. Popular browsers include Microsoft Internet Explorer, Netscape based browsers such as Firefox and Mozilla, and other high quality free browsers such as Opera. All of these browsers work with the web demonstration program that comes with the EtherSmart/WiFi Wildcard (see the demonstration program source code listings at the end of this document).

Additional considerations can limit the performance of some of these browsers if your application needs to serve out more complex web pages that require more than one TCP/IP connection per web page. This can occur, for example, when mixed text and image data originating from the Wildcard are served out in a single web page.

The Lantronix hardware on the Wildcard supports only one active connection at a time. However, the HTTP/1.1 standard (and consequently all browsers in their default configuration) expect the webserver to be able to host two simultaneous connections. A default-configured browser will try to open a second connection when two or more content types (for example, HTML text and a JPEG image) are present in a single web page. The second connection will typically be refused by the Lantronix hardware, causing an incomplete page load. The solution is to configure the browser to expect only one connection from the webserver.

Appendix F explains how to reconfigure Internet Explorer to work with any web page that the EtherSmart/WiFi Wildcard can serve out. For the best solution, though, consider downloading the free Opera browser and using it for all your interactions with the EtherSmart and WiFi Wildcards.

Using Opera Is Highly Recommended

To be able to browse mixed text and graphics pages from the EtherSmart/WiFi Wildcard without modifying your default browser, go to www.opera.com and download the latest version of the Opera browser for Windows desktop machines. It's free, the download file is compact, and the install takes only a few seconds.

Simply go to www.opera.com and select "Download Opera", then double-click on the resulting file to install the browser on your desktop. It is very easy to configure Opera for the EtherSmart/WiFi webserver. Once Opera is installed, simply go to its Tools menu, and select:

Preferences->Advanced->Network->Max Connections Per Server

and enter 1 in the box. Now you're ready to use Opera with the EtherSmart/WiFi Wildcard dynamic webserver. The webserver is described in more detail in a later section.

Using the Free Putty TCP/IP Terminal for Testing and Development

During program development and testing of the EtherSmart/WiFi Wildcard, it is very useful to be able to interact with the Wildcard using a TCP/IP terminal program. A popular option is the "Putty" program, a simple and small freeware program that can be downloaded from the web.

To obtain the free Putty network terminal program, type

Putty

into your search engine (such as Google) and download the executable program from one of the listed sites. It is a simple yet generally useful program.

When you double-click on the Putty desktop icon, a configuration screen appears with a box for the "Host Name or IP address" and another box for the port number. For example, if your EtherSmart or WiFi Wildcard has an IP address of 10.0.1.22 and uses the standard local port value of 80, simply type these values into the appropriate boxes.

A set of radio buttons on the Putty configuration screen allows you to select one of four protocols: "Raw", "Telnet", "Rlogin", and "SSH". The "Raw" mode is a straightforward implementation of a TCP/IP connection, and is recommended for the **`Tunnel_Test`** function in the demo program as described in the section below titled "Serial Tunneling". This test shows how to programmatically control transmission and reception of data over an Ethernet or WiFi link.

"Telnet" is very similar to raw mode, but sends a line of configuration bytes at the start of the session that passes through the Lantronix device to the controller. For this reason, the Raw mode is superior to Telnet for use with the EtherSmart or WiFi Wildcard. The Telnet configuration bytes have the most significant bit set, and can be filtered out by an application program by passing the appropriate flags to routines such as **`Ether_Add_Chars`**, **`Ether_Add_Line`**, **`Ether_Get_Chars`**, **`Ether_Get_Line`**, or by using **`Ether_ASCII_Key`** or **`E_ASCII_Key`**. These functions are discussed in the "Serial Tunneling" section below.

The revectored serial via TCP/IP test as exemplified by the **`Ether_Monitor_Demo`** function in the demonstration code is best conducted using Putty's "Rlogin" mode. The Rlogin protocol mimics the echo and carriage return handling of the standard Mosaic serial terminal, and so is perfect for performing code downloads and interactive testing with Ethernet or WiFi replacing the serial port. This capability is described in the "Code Downloads and Interactive Communications via TCP/IP" section below.

WiFi Access Point Configuration

Figure 1-3 shows a typical wireless networking setup. At the center of the diagram is a wireless access point (AP), also known as a wireless router or bridge. At the left of the diagram is "your" PC, which may be connected to the access point via Ethernet or WiFi. A wireless antenna icon is shown on the access point, with wireless connections to a laptop computer and to the WiFi Wildcard. An optional Ethernet connection from the access point to a WAN (Wide Area Network) with full Internet access is also shown.

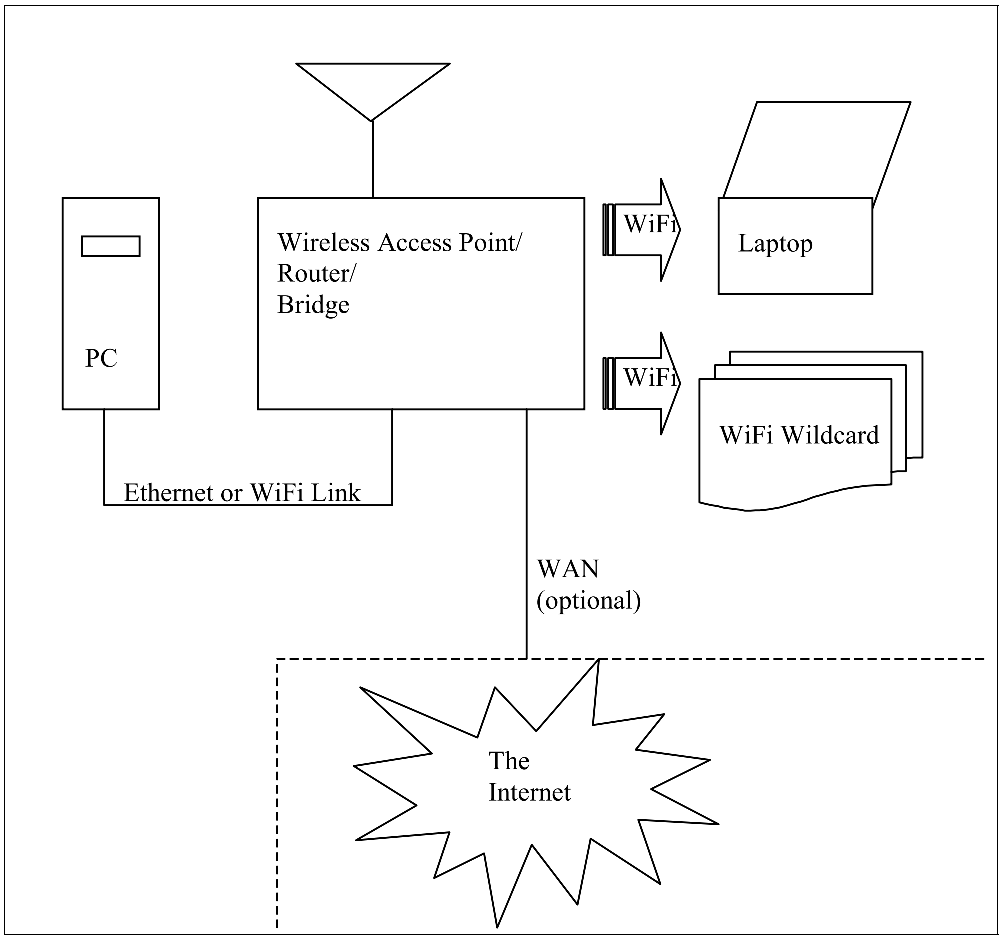


Figure 1-3 Diagram of a typical Wireless Local Area Network (WLAN).

The access point routes network traffic between your PC, the wireless devices such as laptops and the WiFi Wildcard(s), and the optional Internet connection. It is also called a "bridge" because it bridges the PC's network to one or more additional networks, including the WiFi WLAN (Wireless Local Area Network). The access point also contains a DHCP (Dynamic Host Configuration Protocol) server that automatically hands out IP addresses. This is typically how the WiFi Wildcard obtains its IP address.

The wireless access point must of course be configured to operate properly within your network. This section describes how to configure a prototypical access point.

Modern access points include an embedded web server that publishes a configuration web site so you can configure the access point via your PC's browser. In this section we discuss the configuration process for a Netgear Wireless Router (Model WGR614 v4). While your access point may be different, the basic principles discussed here should guide your configuration process. In all cases, be sure to discuss the network configuration issues with your organization's network system administrator.

Printed on the bottom of the Netgear access point are the following key pieces of information:

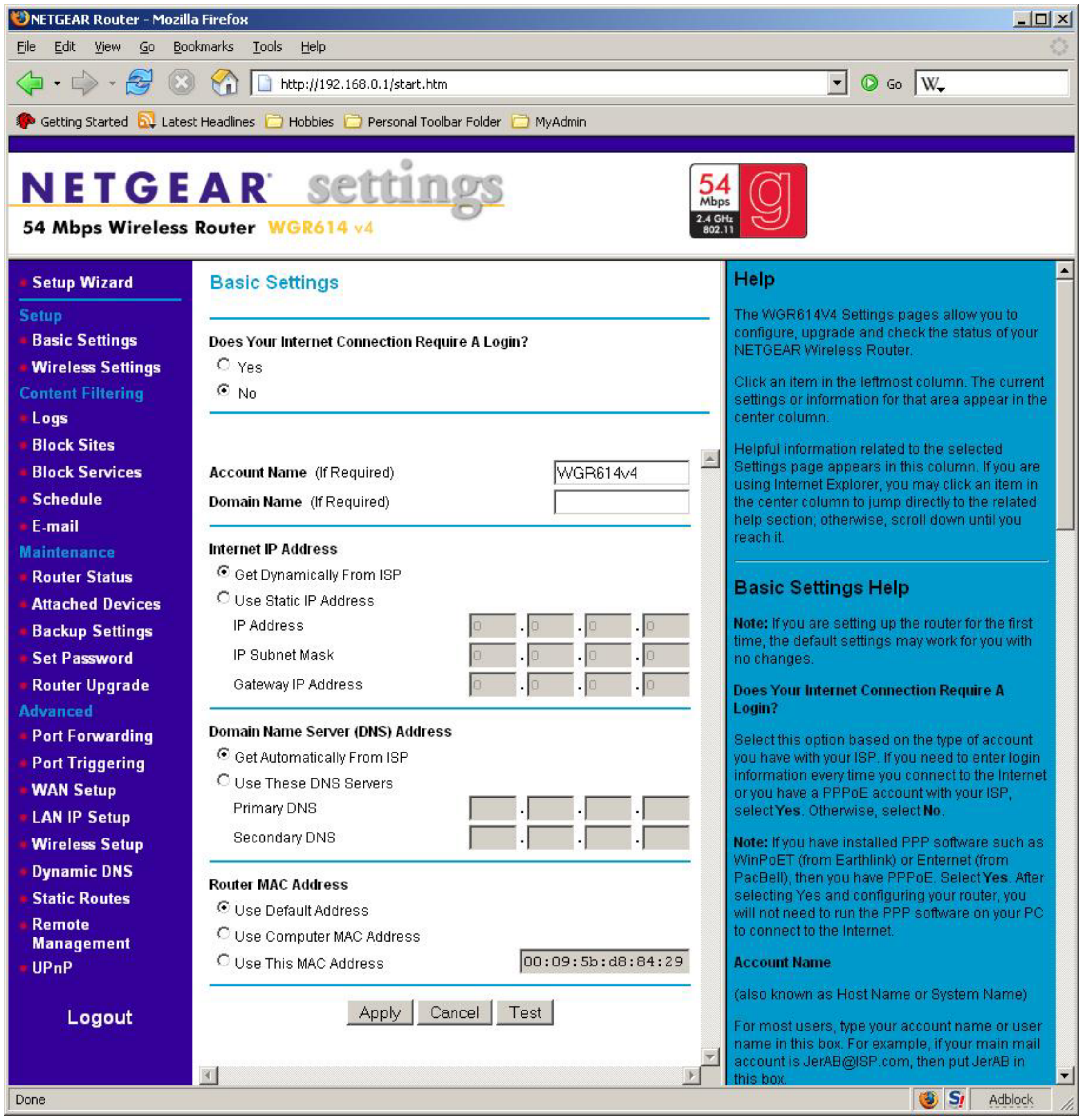
```
DEFAULT ACCESS:
```

```
http://192.168.0.1
user name: admin
password: password
```

To configure the access point, connect it to your PC using a standard Ethernet cable. (It is also possible to connect the access point to your PC using wireless, but this approach is not discussed here). Make sure the access point is powered up, then open your browser, and type into the browser address bar:

```
http://192.168.0.1
```

Type the user name and password into the login box that appears; “admin” and “password” are the defaults in this case. A configuration screen should appear in your browser window. The Netgear home configuration page is shown in Figure 1-4. The available configuration pages are listed in the left hand column. The right hand column presents excerpts from the help files of the router to guide your selections. The “basic settings” displayed in the center of the home page are the default values for the access point. These settings are relevant if the wireless access point is connected directly to the Internet via an ISP (Internet Service Provider). The defaults configure the access point to get its Internet IP and name automatically, and do no harm if the access point is not directly connected to the Internet via an ISP. In this discussion, we’ll assume that the wireless access point is not the main gateway to the Internet.



Basic Settings

Does Your Internet Connection Require A Login? Yes [✓ No]

Account Name (If Required): WGR614v4
Domain Name (If Required):

Internet IP Address: [✓ Get Dynamically From ISP] Use Static IP Address
IP Address: 0.0.0.0
IP Subnet Mask: 0.0.0.0
Gateway IP Address: 0.0.0.0

Domain Name Server (DNS) Address: [✓ Get Automatically From ISP] Use These DNS Servers
Primary DNS:
Secondary DNS:

Router MAC Address: [✓ Use Default Address] Use Computer MAC Address Use This MAC Address: 00:09:5b:d8:84:29

Help

The WGR614V4 Settings pages allow you to configure, upgrade and check the status of your NETGEAR Wireless Router.

Click an item in the leftmost column. The current settings or information for that area appear in the center column.

Helpful information related to the selected Settings page appears in this column. If you are using Internet Explorer, you may click an item in the center column to jump directly to the related help section; otherwise, scroll down until you reach it.

Basic Settings Help

Note: If you are setting up the router for the first time, the default settings may work for you with no changes.

Does Your Internet Connection Require A Login?

Select this option based on the type of account you have with your ISP. If you need to enter login information every time you connect to the Internet or you have a PPPoE account with your ISP, select **Yes**. Otherwise, select **No**.

Note: If you have installed PPP software such as WinPoET (from Earthlink) or Enternet (from PacBell), then you have PPPoE. Select **Yes**. After selecting Yes and configuring your router, you will not need to run the PPP software on your PC to connect to the Internet.

Account Name

(also known as Host Name or System Name)

For most users, type your account name or user name in this box. For example, if your main mail account is JerAB@ISP.com, then put JerAB in this box.

Figure 1-4 Home page of the wireless access point configuration site.

In the left-hand column of Figure 1-4, we click on “Wireless Settings” (the second item in the “Setup” section). This brings up the wireless settings configuration page illustrated in Figure 1-5. The first box in this page is very important: it is the SSID (Service Set IDentifier) of the Wireless Local Area Network (WLAN). The SSID must be shared by all of the devices on the WLAN in order for communications to occur. SSID’s are case sensitive. The default SSID for the WiFi Wildcard is:

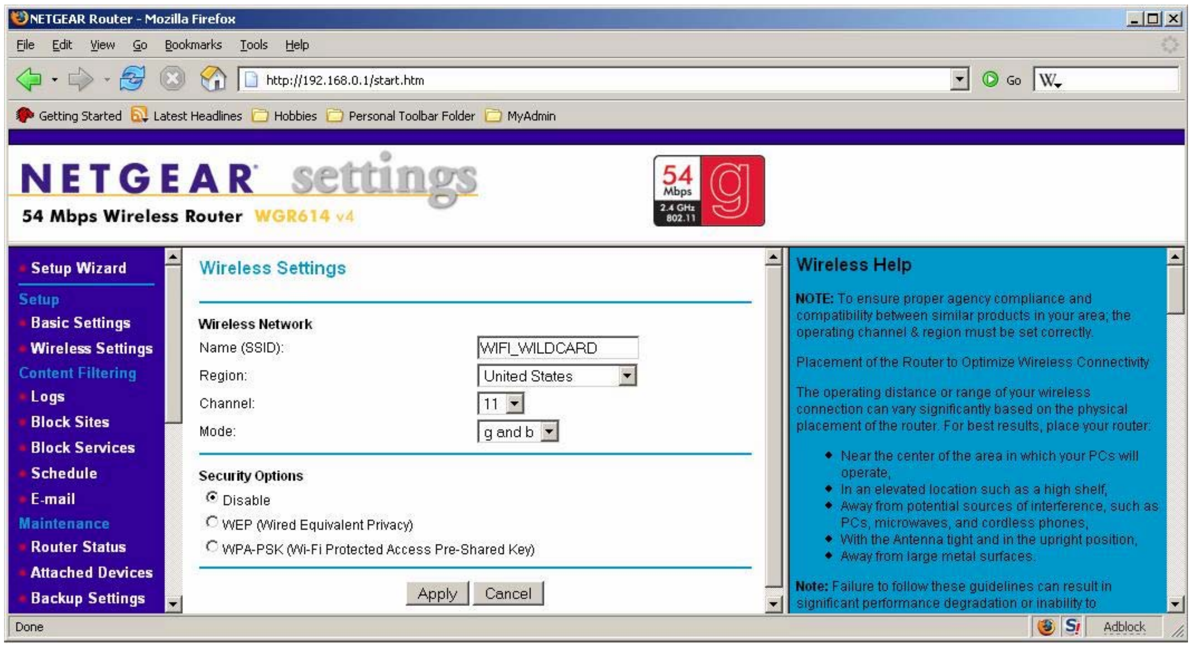
WIFI_WILDCARD

Note that this SSID has been typed into the “Name (SSID)” field in the wireless settings page of the access point configuration site as shown in Figure 1-5. Typing in this SSID and clicking the

"Apply" button at the bottom of the screen configures the wireless access point to use this SSID which enables communications with the WiFi Wildcard. You can also change the SSID of the WiFi Wildcard as explained in the section titled "WiFi Wildcard Configuration". Make sure that the SSID of the Wildcard matches the SSID in the "Name (SSID)" field of the access point configuration page shown in Figure 1-5, and that the SSID of any other devices on the WLAN are set to match the chosen SSID name.

The remaining fields in Figure 1-5 specify the region (United States), default channel (11), and WiFi mode (b/g). The b/g mode specifies the 802.11b/g WiFi protocol, which is the one supported by the WiFi Wildcard and most current WiFi devices.

The "Security Options" section in Figure 1-5 has the "Disable" button selected. Like the SSID, the security settings of the wireless access point must match the settings of all of the wireless devices on the WLAN. Wireless security is disabled by default on the WiFi Wildcard, so selecting the "Disable" button in the Security Options as shown in Figure 1-5 allows communications between the wireless access point and the WiFi Wildcard. You can change the security settings of the WiFi Wildcard as explained in the section titled "The WiFi_Security Function". Make sure that the security settings of the Wildcard match those of the access point and any other devices on the WLAN.



Wireless Settings

Wireless Network

Name (SSID): WIFI_WILDCARD

Region: United States

Channel: 11

Mode: g and b

Security Options

[✓ Disable]

WEP (Wired Equivalent Privacy)

WPA-PSK (Wi-Fi Protected Access Pre-Shared Key)

Wireless Help

NOTE: To ensure proper agency compliance and compatibility between similar products in your area, the operating channel & region must be set correctly.

Placement of the Router to Optimize Wireless Connectivity

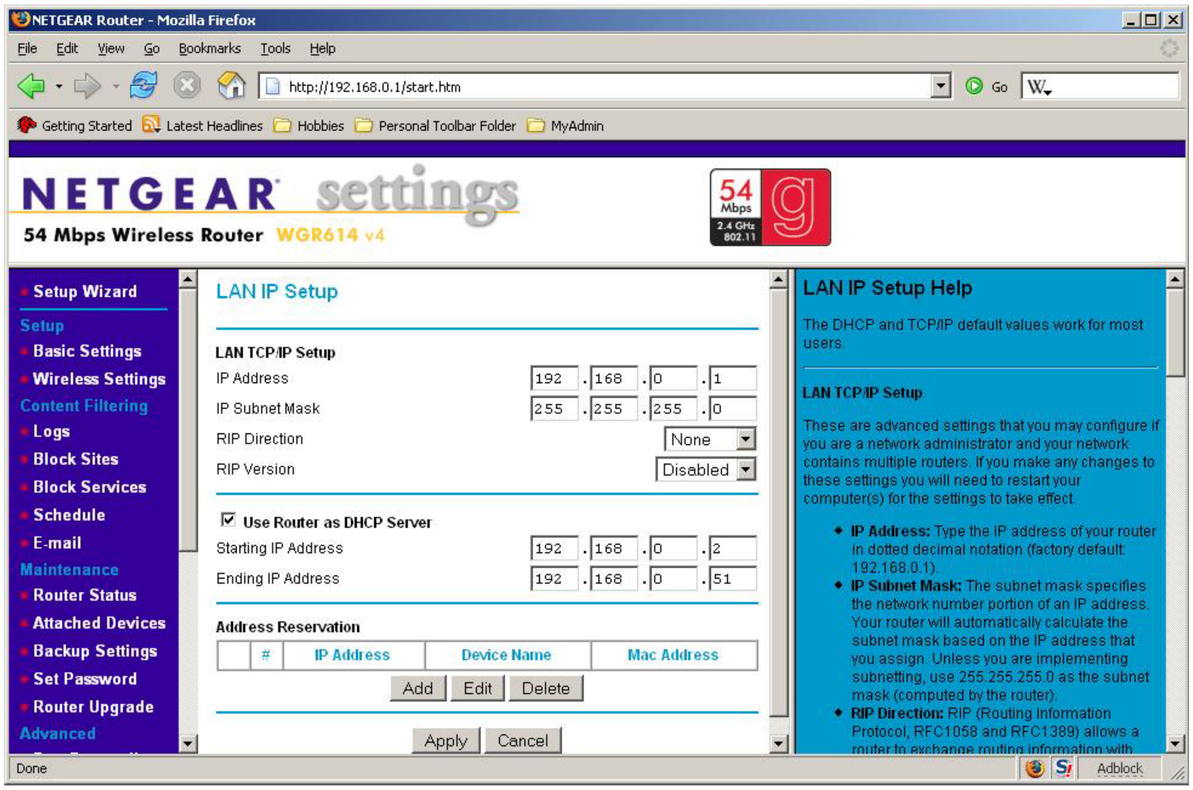
The operating distance or range of your wireless connection can vary significantly based on the physical placement of the router. For best results, place your router:

- Near the center of the area in which your PCs will operate,
- In an elevated location such as a high shelf,
- Away from potential sources of interference, such as PCs, microwaves, and cordless phones,
- With the Antenna tight and in the upright position,
- Away from large metal surfaces.

Note: Failure to follow these guidelines can result in significant performance degradation or inability to

Figure 1-5 Wireless settings page of the access point configuration site.

We can complete the setup of the wireless access point by clicking on the "LAN IP Setup" item in the "Advanced" section of the left hand column of Figure 1-4. The result is the LAN IP configuration page shown in Figure 1-6. This configures the connection between your PC and the access point, and controls whether the access point acts as a DHCP server on the WLAN.



LAN IP Setup

LAN TCP/IP Setup

IP Address	192 . 168 . 0 . 1
IP Subnet Mask	255 . 255 . 255 . 0
RIP Direction	None
RIP Version	Disabled

☑ Use Router as DHCP Server

Starting IP Address	192 . 168 . 0 . 2
Ending IP Address	192 . 168 . 0 . 51

Address Reservation

	#	IP Address	Device Name	Mac Address

Add Edit Delete

Apply Cancel

LAN IP Setup Help

The DHCP and TCP/IP default values work for most users.

LAN TCP/IP Setup

These are advanced settings that you may configure if you are a network administrator and your network contains multiple routers. If you make any changes to these settings you will need to restart your computer(s) for the settings to take effect.

- **IP Address:** Type the IP address of your router in dotted decimal notation (factory default: 192.168.0.1).
- **IP Subnet Mask:** The subnet mask specifies the network number portion of an IP address. Your router will automatically calculate the subnet mask based on the IP address that you assign. Unless you are implementing subnetting, use 255.255.255.0 as the subnet mask (computed by the router).
- **RIP Direction:** RIP (Routing Information Protocol, RFC1058 and RFC1389) allows a router to exchange routing information with

Figure 1-6 LAN IP Setup page of the wireless access point configuration site.

The top set of fields in Figure 1-6 specifies the IP (Internet Protocol) address of the wireless access point itself. Note that the default is 192.168.0.1, which is the same IP address we typed in our browser's address bar to access the configuration pages. If you change this IP address in Figure 1-6 and click "Apply", you will have to type the newly specified IP address in your browser's address bar to access the access point's configuration pages. Changing the IP may be necessary if there is another device on your LAN that has the same IP address, as each IP address must be unique on a given network. By convention, the set of IP addresses starting with 192.168 (as in 192.168.0.1) and the set of IP addresses starting with 10 (as in 10.0.1.22) are reserved for local networks. Consult your network system administrator for advice on specifying the IP address of the wireless access point.

The next set of fields in Figure 1-6 specifies the IP subnet mask. The default is 255.255.255.0. The IP bits that have a corresponding subnet bit of 0 are free to vary, while the IP bits that have a corresponding subnet bit of 1 must match the specified IP address of the access point. In this case, only the last byte of the IP address can vary within the access point's network. For example, valid IP addresses on the WLAN might include 192.168.0.50 or 192.168.0.200, while 192.168.1.50 would be invalid.

The "RIP Direction" and "RIP Version" fields retain their default values of "None" and "Disabled", respectively. RIP applies to larger networks of access points, and is beyond the scope of this document.

The next section of Figure 1-6 configures DHCP (Dynamic Host Configuration Protocol) which can automatically assign IP addresses to devices on the WLAN. By default, the WiFi Wildcard expects to be assigned its IP address by a DHCP server. If there is no other DHCP server available on the network, then the “Use Router as DHCP Server” item in Figure 1-6 should be checked. Note that there can be at most one DHCP server on any portion of the network. In Figure 1-6 the allowed IP addresses to be assigned by the access point are specified to be in the range 192.168.0.2 to 192.168.0.51. Note that these addresses fall within the IP and subnet specification in Figure 1-6.

The “Wireless Setup” page of the Netgear configuration site (in the “Advanced” section of the left hand column in Figure 1-4) simply has two check boxes, both enabled by default: “Enable Wireless Access Point” and “Enable SSID Broadcast”. Both of these should remain checked in most applications.

The wireless access point configuration parameters specified in this section should enable wireless communications with a WiFi Wildcard that is in its default factory condition. Please work with your organization’s network administrator to customize the settings to the needs of your particular network topology.

Verifying WiFi Wildcard Communications

Once you have configured your wireless access point to have an SSID of “WIFI_WILDCARD” (all upper case) with security disabled and the DHCP server enabled, you are ready to verify communications with the WiFi Wildcard in its default factory configuration. (If any of these settings are not compatible with your network topology, you can reconfigure the WiFi Wildcard wireless and security settings as described in later sections of this document).

To test the WiFi Wildcard, it must be powered up and attached to its antenna. Make sure that the WiFi Wildcard is properly mounted on the Wildcard bus, and power up the Mosaic controller board.

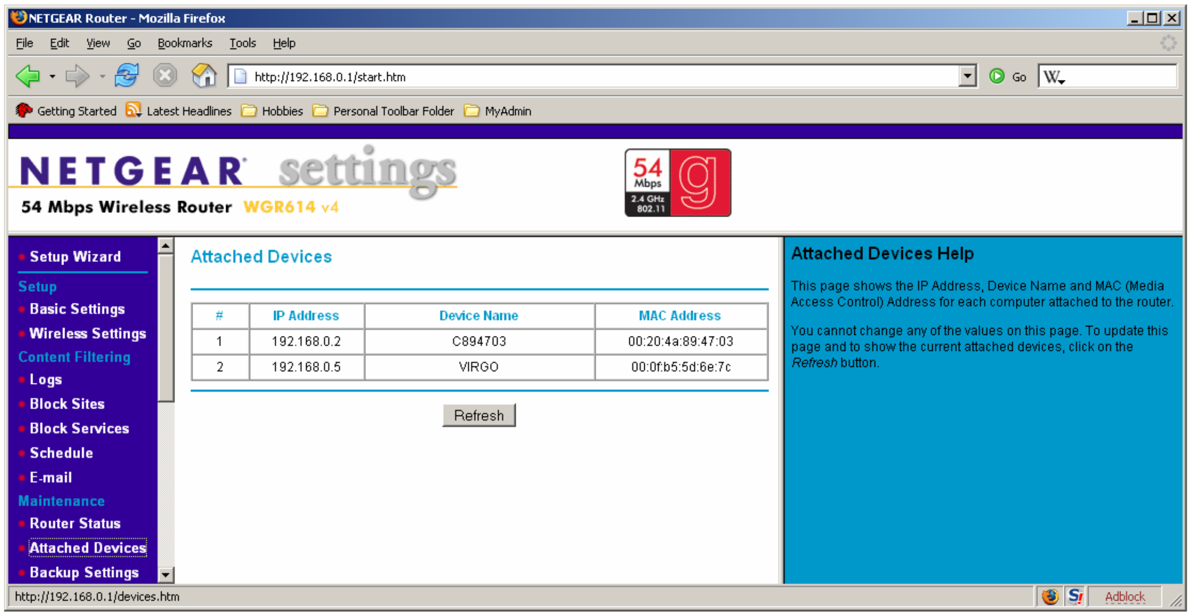
The access point must also be powered up. From the access point’s configuration home page shown in Figure 1-4, click on “Attached Devices” in the “Maintenance” section of the left hand column. You should see a screen similar to Figure 1-7. Each of the devices attached to the wireless access point is listed in a table specifying the device’s IP address, name, and MAC (Media Access Control) address. The MAC address of the WiFi Wildcard is printed on the Lantronix WiPort device. It contains twelve hex digits. You can match this MAC address to an entry in the table; in Figure 1-7 it is the first entry. The default name” that the Lantronix WiPort on the WiFi Wildcard reports comprises the letter “C” followed by the last 6 digits of its MAC address; this is displayed in the middle field of the first line of the table in Figure 1-7. The IP address of the WiFi Wildcard is listed in the first column of the table entry that has the MAC address equal to that printed on the WiPort.

WiFi Wildcard IP Address

Knowing the IP address assigned to the WiFi Wildcard is very important; it allows you to communicate with the Wildcard.

If you see an entry corresponding to the WiFi Wildcard’s MAC address in the “Attached Devices” page of your wireless access point, then you have successfully established communications with the

WiFi Wildcard. If not, try power cycling the WiFi Wildcard and the access point, and verify that the SSID and security settings of the Wildcard match those of the access point.



Attached Devices

#	IP Address	Device Name	MAC Address
1	192.168.0.2	C894703	00:20:4a:89:47:03
2	192.168.0.5	VIRGO	00:0f:b5:5d:6e:7c

Attached Devices Help

This page shows the IP Address, Device Name and MAC (Media Access Control) Address for each computer attached to the router.

You cannot change any of the values on this page. To update this page and to show the current attached devices, click on the *Refresh* button.

Figure 1-7 Attached Devices page of the access point configuration site.

Using the Lantronix Device Installer to Verify Communications

An alternative way to verify communications with the WiFi Wildcard is to use the Lantronix Device Installer program. This approach also works for the EtherSmart Wildcard.

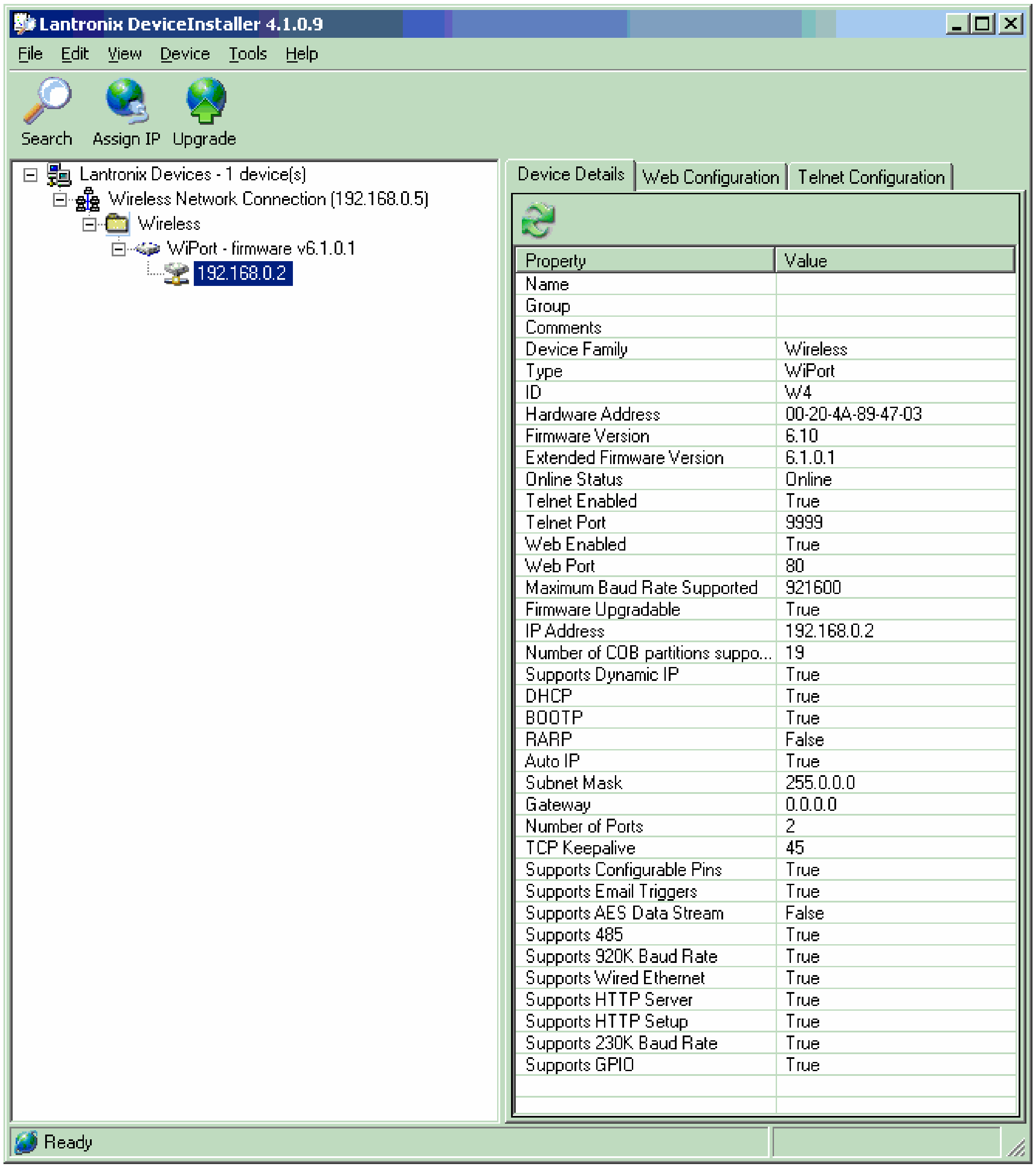
The Lantronix Device Installer is a small application that runs on your PC. It can be downloaded free of charge from the Lantronix web site at www.lantronix.com. Go to the

Support->Downloads -> WiPort G

section of the Lantronix website and download and install the Device Installer (also known as a "configuration utility"). With the access point and the WiFi Wildcard powered up, double click on the Device Installer icon to start it. You should see a screen similar to Figure 1-8 . If not, power cycle the Wildcard and the access point, and activate the Device->Search menu item to rescan the wireless network. If you still are having trouble, go to the Tools->Options->Network menu item and make sure that the "Wireless Network Connection" box is checked.

The Lantronix Device installer provides a list of the IP addresses of all Lantronix devices that are discovered on the specified LAN.

Note that the "Web Configuration" tab does not work with the EtherSmart/WiFi Wildcards, because Mosaic has changed the internal device configuration webserver port to 8000. You can access this website directly from your browser as described in the section titled “Using the Lantronix WiPort Configuration Webserver To Configure Security”.


Lantronix DeviceInstaller 4.1.0.9
File Edit View Device Tools Help
Search
Assign IP
Upgrade
Lantronix Devices - 1 device(s)
Wireless Network Connection (192.168.0.5)
Wireless
WiPort - firmware v6.1.0.1
192.168.0.2
Device Details
Web Configuration
Telnet Configuration
Ready

Property	Value
Name	
Group	
Comments	
Device Family	Wireless
Type	WiPort
ID	W4
Hardware Address	00-20-4A-89-47-03
Firmware Version	6.10
Extended Firmware Version	6.1.0.1
Online Status	Online
Telnet Enabled	True
Telnet Port	9999
Web Enabled	True
Web Port	80
Maximum Baud Rate Supported	921600
Firmware Upgradable	True
IP Address	192.168.0.2
Number of COB partitions suppo...	19
Supports Dynamic IP	True
DHCP	True
BOOTP	True
RARP	False
Auto IP	True
Subnet Mask	255.0.0.0
Gateway	0.0.0.0
Number of Ports	2
TCP Keepalive	45
Supports Configurable Pins	True
Supports Email Triggers	True
Supports AES Data Stream	False
Supports 485	True
Supports 920K Baud Rate	True
Supports Wired Ethernet	True
Supports HTTP Server	True
Supports HTTP Setup	True
Supports 230K Baud Rate	True
Supports GPIO	True

Figure 1-8 Screen shot of Lantronix Device Installer showing the IP address and configuration settings of the WiFi Wildcard.

The EtherSmart/WiFi Software Driver Routines

This section summarizes the capabilities of the EtherSmart/WiFi driver functions. For a complete listing of all of the driver functions, see the "Categorized List of EtherSmart/WiFi Library Functions" in the Glossary. The glossary contains complete descriptions of each function. In this section we review the functions to put them into context.

The driver code contains a suite of functions that make it easy to:

### Initialize and configure the EtherSmart/WiFi Wildcard,

### Perform code downloads and debugging via Ethernet or WiFi,

### Set up serial tunneling connections between two Ethernet- or WiFi -enabled devices,

### Send emails, and,

### Setup a dynamic webserver for instrument monitoring and control.

After a summary of the data structures used by the driver and an introduction to the demonstration program, each of these capabilities is discussed in turn.

EtherSmart/WiFi Driver Data Structures

There are two distinct modes in which the EtherSmart/WiFi Wildcard can be used. One mode is to use the Wildcard to replace the serial port on the controller, enabling code downloads and interactive development to be performed via Ethernet or WiFi instead of via RS-232. This is called "revectored serial via TCP/IP". The other mode is to configure the EtherSmart/WiFi Wildcard to provide communications services including serial tunneling, outgoing email, and web service. These two modes use different data structures as described in the following subsections.

Revectored Serial via TCP/IP

Revectored serial via TCP/IP is conceptually very simple: the three fundamental serial primitives **`Key`**, **`AskKey`**, and **`Emit`** are "revectored" so that they use the Ethernet or WiFi Wildcard port instead of the serial port on the controller. The simplest approach to revectoring is to store the correct modulenum into the global variable named **`ether_revector_module`**, and then to call **`Ether_Serial_Revector`**. This runs the default (startup) task through the specified EtherSmart/WiFi Wildcard with no additional data structures needed. Setting up a separate Ethernet task for revectoring is possible but not required. See the section titled "Code Downloads and Interactive Communications via TCP/IP" below for more details.

Communications Services Require a Dedicated Ethernet Task

Most users will want to configure the EtherSmart Wildcard to provide communications services including serial tunneling, outgoing email, and web service. This requires more sophisticated data structures and a dedicated task to run the services. The pre-coded driver functions automate the required interactions between the application program and the Ethernet task.

Ethernet Task

The "Ethernet task" works for both Ethernet and WiFi communications.

As explained in the User Guide for your Mosaic controller board, a "task" is an environment that is capable of running a program. The task contains a user area, stacks, and buffers that enable a program to run independently of other tasks. The operating system switches between tasks using cooperative multitasking mediated by the Pause routine, and using the timeslice multitasker to force periodic task switches. Each of the tasks typically manages a defined set of services, and the tasks interact by means of shared variables or mailboxes.

Mailboxes

A "mailbox" is a 32-bit variable in common memory that conveys a message from one task to another. A mailbox must be cleared before a new message may be sent; this helps to synchronize the tasks and ensures that no messages are discarded or written over without being read by the receiving task. A mailbox may be read and cleared by a "blocking" function that waits until the mailbox contains a message (that is, until it contains a non-zero value) and then reads out, reports, and zeros the mailbox. It may also be checked by a "non-blocking" function that reads and reports the contents of the mailbox and, if it contains a non-zero value, clears it. Non-blocking mailbox reads are often preferred because they avoid tying up one task while waiting for another task to send a message.

The EtherSmart/WiFi driver code uses mailboxes to coordinate the provision of communications services by the Ethernet task to the programmer's main application task. This is all done transparently, so you don't have to be an expert on inter-task communications to use the EtherSmart/WiFi Wildcard. Three 32-bit mailboxes named **`ether_command`**, **`ether_response`**, and **`ether_gui_message`** are allocated in common RAM when the initialization routine executes. For example, when the application program invokes a command such as **`Ether_Send_Email`**, a message is dispatched via the **`ether_command`** mailbox to the Ethernet task. Consequently, the Ethernet task opens a connection to the specified remote computer, sends the email, and closes the connection. The Ethernet task then sends a result error code in the **`ether_response`** mailbox. This mailbox must be cleared by the application program using either the blocking function **`Ether_Await_Response`**, or the non-blocking function **`Ether_Check_Response`**. The main loop of your program's application task should periodically call one of these two functions to manage the interactions with the Ethernet task. See the Email and Serial Tunneling sections (and the corresponding functions in the demo program) for examples of use.

Variables

In addition to the mailboxes, there are two variables declared in the driver code named **`ether_revector_module`** and **`ether_service_module`**. As mentioned above, **`ether_revector_module`** specifies which module (if any) is being used for revectored serial via TCP/IP (Ethernet or WiFi). The **`ether_service_module`** variable is set by the initialization functions **`Ether_Init`** or **`WiFi_Init`** and their calling functions (**`Ether_Setup`**, **`Ether_Setup_Default`**, and **`Ether_Task_Setup`**, or **`WiFi_Setup`**, **`WiFi_Setup_Default`**, and **`WiFi_Task_Setup`**). The **`ether_service_module`** variable specifies which Wildcard module is accessed by the **Ether_Service_Loop** routine running in the Ethernet control task. Because each of these variables is automatically initialized by higher level functions, you typically don't have to worry about them. It is easy to install two EtherSmart and/or WiFi Wildcards, one for revectored serial, and the other providing communications services including serial tunneling, email, and web service.

If you need more than one Wildcard providing TCP/IP communications services, consult the section below titled "Initializing Multiple EtherSmart/WiFi Wildcards".

Information Structure Holds Control Parameters

Most of the control data for the EtherSmart/WiFi Wildcard is contained in a structure whose extended 32-bit base xaddress is returned by the **Ether_Info** function. There is one of these structures for each initialized EtherSmart/WiFi Wildcard providing communications services. This structure contains error results, flags to enable the web server and serial tunneling services, buffer sizes, pointers to the mailboxes and buffers, web service form parsing pointers, configuration data, and security parameters. The configuration data includes local and gateway IP addresses, ports and netmasks, timeout parameters, and pointers to strings used by the email routine. The structure also stores the input parameters passed to each service request function until the Ethernet task can extract the parameters and perform the requested action.

Several of the functions in the driver code allow the programmer to set and read the values in the **Ether_Info** structure. The structure is declared and initialized by the functions **Ether_Init** or **WiFi_Init** and their calling functions (**Ether_Setup**, **Ether_Setup_Default**, and **Ether_Task_Setup**, or **WiFi_Setup**, **WiFi_Setup_Default**, and **WiFi_Task_Setup**). These functions write reasonable default values into the this data structure.

Allocated Buffers Support Tunneling, Email and Web Services

Communicating via TCP/IP over Ethernet or WiFi requires the use of buffers to hold incoming and outgoing data. The input and output buffers for serial tunneling and email are called **Ether_Inbuf** and **Ether_Outbuf**, respectively. The web service input and output buffers are called **HTTP_Inbuf** and **HTTP_Outbuf**. Each of these buffers has a default maximum size that is set at initialization time. The default size of **Ether_Inbuf** and **Ether_Outbuf** is given by the constant **ETHER_BUFSIZE_DEFAULT**. The default size of the **HTTP_Inbuf** and **HTTP_Outbuf** are given by the constants **HTTP_INBUFSIZE_DEFAULT** and **HTTP_OUTBUFSIZE_DEFAULT**, respectively. The functions that write to the buffers typically accept the maximum buffer size as an input parameter, and ensure that they never write beyond the allowed buffer size.

If you need specify a new buffer base xaddress or size that is different from the values set at initialization time, use the functions **Ether_Set_Inbuf**, **Ether_Set_Outbuf**, **HTTP_Set_Inbuf**, and **HTTP_Set_Outbuf**.

The LBuffer, a Counted Buffer

Functions that manipulate buffers need to know the number of valid bytes stored in the buffer. A convenient place to store this information is in the buffer itself. For this reason, this driver code uses a data structure called an "LBuffer", defined as a buffer with a 16-bit count stored in the first two bytes of the buffer, with the data following. For example, a function like **Ether_Send_LBuffer** expects the 32-bit extended address (xaddress) of the LBuffer as an input parameter, and sends the contents on the active TCP/IP connection. This function automatically fetches the number of data bytes in the buffer (its "count") from the buffer's first 2 bytes, and sends data starting at the specified xaddress+2. Data reception functions such as **Ether_Get_Data**, **Ether_Get_Chars** and **Ether_Get_Line** expect the 32-bit extended address (xaddress) of an LBuffer as an input parameter, and receive incoming data to the buffer, storing the count at the specified xaddress, and storing the data starting at xaddress+2. The glossary entry for each buffer handling function tells whether it is dealing with an uncounted buffer or an LBuffer.

Note that the maximum size of a buffer set at initialization time refers to the data portion of the buffer; the allocated size of the LBuffer is two bytes larger than the data size. For example, **ETHER_BUFSIZE_DEFAULT** equals 510 bytes. The resulting allocated size of **Ether_Inbuf** and **Ether_Outbuf** is 512 bytes each, with the first 2 bytes of each buffer containing the count, and the remaining 510 bytes containing the data.

Web Handlers Are Posted to the Autoserve Array

The EtherSmart/WiFi Wildcard driver includes a programmable dynamic webserver. You configure the webserver by defining and posting a "handler function" for each web address that is implemented. When the browser requests the web address (known as a URL, or Uniform Resource Locator), the webserver automatically executes the associated handler which performs any required actions and transmits the web page to the browser. As described in a later section, versatile functions make it easy to use HTML "forms" to request particular data or stimulate particular actions from the Mosaic controller.

The "Autoserve Array" is a data structure that holds a pointer to each URL string, and the corresponding 32-bit function pointer (execution address) of the handler associated with that URL. **HTTP_Autoserve_Ptr** returns an address that contains the 32-bit base address of this array which is set up by the functions **Ether_Init** or **WiFi_Init** and their calling functions (**Ether_Setup**, **Ether_Setup_Default**, and **Ether_Task_Setup**, or **WiFi_Setup**, **WiFi_Setup_Default**, and **WiFi_Task_Setup**). The functions **HTTP_Add_Handler** and **HTTP_Add_GUI_Handler** post the required information into this array, so the programmer does not need to directly access the contents of this low-level data structure.

Passing String Extended Addresses as Function Parameters

A number of the functions in the EtherSmart/WiFi software driver expect as input parameters the 32-bit extended address ("xaddress") of a string. An xaddress includes page information that is not available in a 16-bit address. Because by default the C compiler treats string addresses as 16-bit parameters, a conversion must be performed to generate the 32-bit xaddress. Listing 1-1 from the **strdefs.h** file referenced by a **#include** statement in the C demonstration (demo) program illustrates a macro called **STRING_XADDR** that performs the required conversion. Simply invoke the **STRING_XADDR** macro with the string specifier as the macro argument to generate the corresponding 32-bit xaddress. It is best to use the macro right in the argument list of the function being called. For an example of use, see Listing 1-8.

Listing 1-1 STRING_XADDR Macro Converts a String to an Xaddress.

```
// ************** USEFUL MACRO FOR STRINGS IN V4.xx C COMPILER **************
// the TO_XADDR defined in /mosaic/include/types.h
// transforms a separate 16-bit addr, page into a 32-bit xaddress:
// #define TO_XADDR(address,page)     ((xaddr) (((page)<<16)+ (0xFFFF & (address))))
// We want to substitute THIS_PAGE (also defined in types.h) for the page,
// as the V4.xx C compiler replicates the strings on each page;
// therefore, in most cases, the calling function's page
// is the same as the string page:
#define STRING_XADDR(str_addr) ((xaddr) TO_XADDR(((xaddr) str_addr), \
         ((xaddr) THIS_PAGE)))
```

The Demo Program Provides an Example of Use

For an example of an application program that exercises the capabilities of the EtherSmart Wildcard, see the demonstration ("demo") program listings that are provided in both C and Forth in Appendices B and D of this document. Appendix A tells you how to compile and load the program. Briefly, you'll need a version of the pre-compiled driver code which includes an install file containing the code in S-record format, and `*.c` and `*.h` (C code and header) files.

You'll also need the "resource files" created by the "Image Converter" program that is part of the CD you get from Mosaic when you buy your starter kit. The Image/Resource converter creates an `image_data.txt` download file and an `image_headers.h` (for C) and `image_headers.4th` (for Forth) header file that encode any images and/or HTML pages you define if you are implementing a web server. After sending the driver's install file and the `image_data.txt` file to the controller, you'll send the program. C programs are compiled by clicking the "make" icon in the Mosaic IDE and sending the resulting file to the controller. Forth programs are compiled by simply sending the source file to the controller using the QED Terminal program. Note that the demo program references the driver library files using `#include` statements near the top of the file listing.

The remainder of this section presents code examples that are excerpted from the C version of the demo program. We recommend that you look at the code listing now in Appendix B or D to familiarize yourself with the demo program. Because function names and parameter lists are the same in both the C and Forth versions of the code, interested Forth programmers can look up the relevant excerpts by examining Appendix D and locating the referenced function name.

Recall that most of the functions expect the Wildcard address called the "module number" or simply "modulenum" in their input parameter lists. Make sure that the jumper settings on the Wildcard hardware match the modulenum parameter that is specified in your application program. In the demo program, the constant `E_MODULENUM` is declared to hold the modulenum. Its default value is 4 in the provided code. If your EtherSmart or WiFi Wildcard is not installed as module number 4, you must change the value of this constant to equal the hardware modulenum as specified in the "Selecting the Wildcard Address" section above.

Loading Your Program

See Appendix A for a discussion of how to load the resources and the kernel extension library files onto the controller board before loading your custom application program. Appendix A also summarizes how to get the IP address needed to exercise the web demonstration pages.

Once you've loaded the demo program onto your controller board as described in Appendix A and typed `Ether_Web_Demo` or `WiFi_Web_Demo` to start the program, you can learn the EtherSmart or WiFi Wildcard's IP address by typing at the QED Terminal:

```
Ether_IP_Report(  )
```

(Forth users would type the `Ether_IP_Report` command without any parentheses).

Assuming that the reported IP address is non-zero, you can then type the reported IP address into the web address bar of your web browser to see the demonstration web page. If the reported IP address is 0.0.0.0, then there is no DHCP service running on your LAN to assign an IP address. See the section below titled "Assigning an IP Address to the EtherSmart or WiFi Wildcard".

The remainder of this document explains all the features of the demo program.

Initialization, Configuration and Diagnostics

The pre-coded driver contains a suite of functions that make it easy to initialize and configure the EtherSmart and WiFi Wildcards. The relevant functions are summarized in the EtherSmart/WiFi Wildcard Glossary document. For a list of functions, consult the "Configuration and Diagnostics", "Initialization", and "WiFi Configuration" tables in the "Categorized List of EtherSmart/WiFi Library Functions" in the Glossary. The glossary contains complete descriptions of each function. In this section we review the functions to put them into context.

Initialization Functions

The EtherSmart and WiFi Wildcards share the same code base, but are distinguished at initialization time by the functions listed in Table 1-3. The low level function `WiFi_Module` identifies a specified module as a WiFi Wildcard. For each of the EtherSmart initialization functions that start with the `Ether_` prefix, a parallel initialization function that starts with the `WiFi_` prefix is available to initialize the WiFi Wildcard. The latter functions include a call to the `WiFi_Module` function. The last row in Table 1-3 summarizes the calling order of the initialization functions. Consult the function glossary entries for detailed descriptions of each function.

The initialization process marks the identity of the Wildcard. For example, the `Ether_Setup` function initializes the EtherSmart Wildcard using specified memory parameters, while the parallel function `WiFi_Setup` initializes the WiFi Wildcard. In fact, `WiFi_Setup` simply calls `Ether_Setup`, and then invokes the low level `WiFi_Module` utility function to declare the specified module as a WiFi Wildcard.

Table 1-3 Initialization functions for EtherSmart and WiFi Wildcards.

EtherSmart Wildcard Init Functions	WiFi Wildcard Init Functions
Ether_Info_Init	**WiFi_Info_Init**
Ether_Init	**WiFi_Init**
Ether_Setup	**WiFi_Setup**
Ether_Setup_Default	**WiFi_Setup_Default**
Ether_Task_Setup	**WiFi_Task_Setup**
	WiFi_Check
	WiFi_Module
Ether_Task_Setup calls Ether_Setup_Default calls Ether_Setup calls Ether_Init calls Ether_Info_Init.	WiFi_Task_Setup calls WiFi_Setup_Default calls WiFi_Setup calls WiFi_Init calls WiFi_Info_Init.

EtherSmart Wildcard Initialization Functions

For the EtherSmart Wildcard, the highest level initialization function in the pre-compiled device driver library is **Ether_Task_Setup** which accepts as input parameters a 16-bit **TASK** base address in common memory and the modulenum. (For the definition of the **TASK** structure, see the **user.h** file in the Mosaic include directory of your C compiler distribution.) Listing 1-2 shows an example of a function called **Ether_Task_Setup_Default** from the demo program that invokes **Ether_Task_Setup**, passing it the base address of the **ether_control_task**, and the **E_MODULENUM** Wildcard address.

Listing 1-2 Using EtherTaskSetup_Default() from the Demo Program to Initialize the EtherSmart Wildcard Services.

```
// ****** SETUP ETHERNET TASK TO RUN ETHERSMART WEB AND TUNNELING SERVICES ******

TASK ether_control_task;  // 1 Kbyte per task area

void Ether_Task_Setup_Default( void )
// performs full initialization of the ether_info struct and mailboxes for the
// modulenum specified by the E_MODULENUM constant, and
// builds and activates an ethernet control task
// to service the EtherSmart Wildcard
{
 // NEXT_TASK = TASKBASE;  // empties task loop; comment out if other tasks allowed
   Ether_Task_Setup( &ether_control_task, E_MODULENUM);
}
```

Ether_Task_Setup is a high level routine that performs a full initialization of the **Ether_Info** structure and mailboxes, and builds and activates an Ethernet control task to service the Wildcard with the specified modulenum. This function calls a chain of lower level initialization functions, each of which is available for use if the default high level routines do not meet the needs of your application. **Ether_Task_Setup** builds and activates a task running the **Ether_Service_Loop** function, and calls **Ether_Setup_Default. Ether_Setup_Default** specifies a default buffer area base (0x043000 for V4.xx platforms, and 0x178000 for V6.xx platforms) and calls **Ether_Setup**. About 3 Kbytes is allocated in this buffer area; see the glossary entry of

Ether_Setup for a detailed description of operation. **Ether_Setup** calls **Ether_Init** with a set of default buffer specifications and sizes. **Ether_Init** calls the lowest level initialization routine **Ether_Info_Init**. Most users of the EtherSmart Wildcard will call only the highest level **Ether_Task_Setup** routine, or its caller in the demo program, **Ether_Task_Setup_Default**. The lower level initialization functions listed in this paragraph allow "power users" to tailor the memory map and/or buffer sizes to suit specialized needs. Consult the EtherSmart/WiFi Glossary for more details about the EtherSmart Wildcard initialization functions.

WiFi Wildcard Initialization Functions

For the WiFi Wildcard, the highest level initialization function in the pre-compiled device driver library is **WiFi_Task_Setup** which accepts as input parameters a 16-bit **TASK** base address in common memory and the modulenum. (For the definition of the **TASK** structure, see the **user.h** file in the Mosaic include directory of your C compiler distribution.) Listing 1-3 shows an example of a function called **WiFi_Task_Setup_Default** from the demo program that invokes **WiFi_Task_Setup**, passing it the base address of the **ether_control_task**, and the **E_MODULENUM** Wildcard address.

Listing 1-3 Using WiFi_TaskSetup_Default() from the Demo Program to Initialize the WiFi Wildcard Services.

```
// ******* SETUP ETHERNET TASK TO RUN WIFI WEB AND TUNNELING SERVICES **********

TASK ether_control_task;  // 1 Kbyte per task area

void WiFi_Task_Setup_Default( void )
// performs full initialization of the ether_info struct and mailboxes for the
// modulenum specified by the E_MODULENUM constant, and
// builds and activates an ethernet control task to service the
// WiFi Wildcard. marks the E_MODULENUM module as a WiFi Wildcard
{
 // NEXT_TASK = TASKBASE;  // empties task loop; comment out if other tasks allowed
   WiFi_Task_Setup( &ether_control_task, E_MODULENUM);
}
```

WiFi_Task_Setup is a high level routine that performs a full initialization of the **Ether_Info** structure and mailboxes, and builds and activates an Ethernet control task to service the WiFi Wildcard with the specified modulenum. This function calls a chain of lower level initialization functions, each of which is available for use if the default high level routines do not meet the needs of your application. **WiFi_Task_Setup** builds and activates a task running the **Ether_Service_Loop** function, and calls **WiFi_Setup_Default**. **WiFi_Setup_Default** specifies a default buffer area base (0x043000 for V4.xx platforms, and 0x178000 for V6.xx platforms) and calls **WiFi_Setup**. About 3 Kbytes is allocated in this buffer area; see the glossary entry of **WiFi_Setup** for a detailed description of operation. **WiFi_Setup** calls **WiFi_Init** with a set of default buffer specifications and sizes. **WiFi_Init** calls the lowest level initialization routine **WiFi_Info_Init**. Most users of the WiFi Wildcard will call only the highest level **WiFi_Task_Setup** routine, or its caller in the demo program, **WiFi_Task_Setup_Default**. The lower level initialization functions listed in this paragraph allow "power users" to tailor the memory map and/or buffer sizes to suit specialized needs. Consult the EtherSmart/WiFi Glossary for more details about the WiFi Wildcard initialization functions.

Configuring the Lantronix Device

The low level Ethernet interface on the EtherSmart Wildcard is provided by the Lantronix XPort, and the low level WiFi interface on the WiFi Wildcard is provided by the Lantronix WiPort. Each of these modular devices combines a processor, flash and RAM memory, network interface controller, and physical interface into a single component. Each Lantronix device can support one Ethernet connection at a time, and limits incoming connections to a single TCP/IP port number that must be specified in advance. Changing the configuration parameters of the Lantronix device is time consuming (approximately 13 seconds) and modifies flash in the device.

The two fundamental functions that configure the Lantronix device on the EtherSmart or WiFi Wildcard are called **`Ether_XPort_Defaults`** and **`Ether_XPort_Update`**. Despite the names, each function works on both the EtherSmart Wildcard and the WiFi Wildcard.

`Ether_XPort_Defaults` sets the recommended default parameters on the EtherSmart or WiFi Wildcard to be the same as those set at the Mosaic factory. Note that these default values are not the same as those set by the Lantronix configuration webserver's "Set Defaults" function. Mosaic's defaults as initialized by **`Ether_XPort_Defaults`** or **`Ether_XPort_Update`** must be set for proper operation of the EtherSmart/WiFi Wildcard. For example, communications between the Wildcard's UART chip and the Lantronix device take place at 115 Kbaud, while the Lantronix default baud rate is only 9600 baud. Execution of **`Ether_XPort_Defaults`** takes approximately 13 seconds and modifies the Lantronix flash, so this routine should not be included in an autostart or **`main`** routine that runs each time the controller powers up.

The Lantronix configuration functions called **`Ether_XPort_Defaults`** and **`Ether_XPort_Update`** can be called interactively during program development and when the instrument is being prepared for shipment. These functions modify flash memory in the Lantronix device. Because flash memory can typically be written only 1000 to 10000 times before burning out, these flash-modifying routines should not be called frequently by a running application program.

A number of convenient interactive configuration, reporting and diagnostic functions are defined in the demo program. Listing 1-4 presents these functions. The **`_Q`** tag marks each function as callable from the terminal monitor. As explained in the Software User Guide for the controller product that you are using, you must type the function name and **`(`** opening parenthesis with no intervening spaces, then at least one space after the **`(`** before any parameters or the closing **`)`** character. If you are typing numeric input parameters, be sure to set the numeric base by typing either **`DECIMAL`** or **`HEX`**. The selected base is retained until changed, so you need only type it once per session.

Before calling any of these functions, be sure to initialize the Wildcard by invoking an initialization function. If you have an EtherSmart Wildcard, you can type one of the following at the terminal prompt:

```
Ether_Task_Setup_Default(  )
```

or:

```
Ether_Web_Demo(  )
```

or any other function that invokes `Ether_Task_Setup_Default`.

If you have a WiFi Wildcard, you can initialize it by typing one of the following at the terminal prompt:

```
WiFi_Task_Setup_Default(  )
```

or:

```
WiFi_Web_Demo(  )
```

or any other function that invokes `WiFi_Task_Setup_Default`.

Listing 1-4 Interactive Configuration, Reporting and Diagnostic Routines Defined in the Demo Program. These work for both EtherSmart and WiFi Wildcards.

```
// ***************** INTERACTIVE CONFIGURATION AND REPORTING ************

_Q int Ether_Set_Defaults( void )
// works for xport or wiport (ethernet/wifi)
// call this AFTER calling main() or Ether_Web_Demo or Ether_Task_Setup_Default()
// or WiFi_Web_Demo or WiFi_Task_Setup_Default.
// sets mosaic factory defaults; returns error code
// sets local IP and gateway to 0.0.0.0 = unassigned, so IP address
// gets assigned via DHCP (Dynamic Host Configuration Protocol) by the LAN's gateway.
// see user guide for more information.
{  printf("\rSetting defaults...\r");
   Ether_XPort_Defaults(E_MODULENUM);
   return((int) Ether_Await_Response(E_MODULENUM)); // error code is in lsword
}

_Q int Ether_Set_Local_IP(int my_ip1, int my_ip2, int my_ip3,  int my_ip4)
// call this AFTER calling main() or Ether_Web_Demo or Ether_Task_Setup_Default()
// or WiFi_Web_Demo or WiFi_Task_Setup_Default.
// sets the IP address of the EtherSmart Wildcard specified by E_MODULENUM as:
//   ip1.ip2.ip3.ip4
// For example, to set the IP address to 10.0.1.22,
// pass to this function the parameters:
//   10 0 1 22
// returns error code
// NOTES: type DECIMAL at the monitor before invoking this function interactively!
//        The input types are declared as int to simplify interactive calling,
//        as the interactive debugger would require char specifiers before each input
//        parameter if the char type were used.
// NOTE: assigning a nonzero IP address disables DHCP!
{  printf("\rSetting local IP address...\r");
   Ether_Local_IP(my_ip1, my_ip2, my_ip3,  my_ip4, E_MODULENUM);
   Ether_XPort_Update(E_MODULENUM);
   return((int) Ether_Await_Response(E_MODULENUM)); // error code is in lsword
}

_Q void Ether_IP_Report(void)
// call this AFTER calling main() or Ether_Web_Demo or Ether_Task_Setup_Default()
// or WiFi_Web_Demo or WiFi_Task_Setup_Default.
// takes 7 seconds to execute, so be patient.
// Report is of the form:
// IP 010.000.001.019 GW 010.000.001.022 Mask 255.255.255.000
// which summarizes the IP address, gateway address, and netmask, respectively.
{  Ether_IP_Info_Report(E_MODULENUM);
```

```
    }
```

The interactive function `Ether_Set_Defaults( )` reverts to the Mosaic factory default setting; it works for both EtherSmart and WiFi Wildcards. It calls `Ether_XPort_Defaults` and waits for the response from the Ethernet task by invoking `Ether_Await_Response`. To use it, simply type from the QED Terminal:

```
Ether_Set_Defaults(  )
```

As noted in the text above and in the comments in the listing, `Ether_XPort_Defaults` works only after the Wildcard has been initialized. `Ether_XPort_Defaults` `SEND`s a message via the `ether_command` mailbox to the task running the `Ether_Service_Loop` which dispatches the action function. This function enters the Lantronix monitor mode, and sends the default versions of flash block0 (serial and network settings), block3 (additional network settings), and block7 (hardware handshaking configuration) S-records. If the Wildcard has been declared as WiFi, block 8 (WiFi configuration settings) and block 9 (WiFi security password) are also initialized. This routine then executes the Lantronix monitor mode reset command to instantiate the new values in Lantronix flash memory. After calling this routine the application must clear the `ether_response` mailbox using `Ether_Check_Response` or `Ether_Await_Response`, but note that the result will not be present until over 13 seconds have elapsed, so please be patient. Consult the glossary entry for implementation details.

If you want to change some of the Mosaic default parameters, use `Ether_XPort_Update` after setting the desired parameter values. The functions listed in Table 1-4 let you change the configurable values that are to be stored in the Lantronix flash:

Table 1-4 Functions that set configurable parameters to be stored by Ether_XPort_Update.

`Ether_DHCP_Name`	`Ether_Remote_IP_Ptr`
`Ether_Gateway`	`Ether_Remote_Port_Ptr`
`Ether_Gateway_IP_Ptr`	`Ether_TCP_Control`
`Ether_Internal_Webserver_Port`	`Ether_Telnet_Password`
`Ether_Local_IP`	`WiFi_Options`
`Ether_Local_Port`	`WiFi_Security`
`Ether_My_IP_Ptr`	`WiFi_SSID`
`Ether_Netmask_Ptr`	

You can invoke one or more of these functions to set parameters, then use `Ether_XPort_Update` to instantiate the values in the Lantronix flash. If none of the configuration functions shown in Table 1-4 has been executed, then `Ether_Xport_Update` performs the same action as `Ether_XPort_Defaults`. For a WiFi Wildcard, all of the functions in Table 1-4 are relevant. For an EtherSmart Wildcard, the `WiFi_Options`, `WiFi_Security`, and `WiFi_SSID` functions have no effect.

The functions that end in **_Ptr** return a 32-bit pointer xaddress in RAM that can be used to read or write the associated value. The other functions in Table 1-4 accept parameters and write the new values into the **Ether_Info** structure for use by **Ether_XPort_Update**; see their glossary entries for more details. Be careful before changing the default values. For example, changing the **Ether_Local_Port** to a number other than its default value of 80 will impact the dynamic webserver, requiring browsers to specify a different port than the default when accessing the EtherSmart/WiFi dynamic webserver. (Typing the IP address followed by a colon and then the port number in the browser's address bar allows a browser to connect to a webserver on the specified port).

The default values associated with these parameters are as follows. The **Ether_Internal_Webserver_Port** (for the configuration by web feature) is 8000. By default, there is no DHCP name or telnet password assigned. See the glossary entry for **Ether_TCP_Control** for a description of the default values of the associated low-level timeouts and character handling parameters. The **Ether_Gateway**, **Ether_Local_IP**, **Ether_Remote_IP**, and netmask are all set to 0.0.0.0 (unassigned). Note that the **Ether_Remote_IP** parameter is never used by the driver software, as all connections explicitly specify the remote IP and port as a set of passed parameters. The ability to change the remote IP value stored in Lantronix flash is provided only for the sake of completeness.

See the glossary entry of **WiFi_Options** for a description of the transmit power and rate settings, and the "ad hoc" mode; these are also described in detail in the WiFi Configuration section. The default **WiFi_Security** settings turn off all security and encryption. The default **WiFi_SSID** (Service Set ID) is "WIFI_WILDCARD" as described in the WiFi Configuration section.

Assigning an IP Address to the EtherSmart or WiFi Wildcard

The Mosaic factory defaults set via **Ether_XPort_Defaults** set the local IP address and the Gateway IP address and netmask to 0.0.0.0. This has the effect of causing the Wildcard to wait for an IP address to be assigned via DHCP (Dynamic Host Configuration Protocol). The DHCP server that assigns the IP is typically running on the gateway computer of the LAN (Local Area Network), or the gateway or wireless access point of the WLAN (Wireless Local Area Network). If your LAN or WLAN has an active DHCP service, then the EtherSmart/WiFi Wildcard should be assigned an IP address automatically when it is powered up and connected to the network. (See the "WiFi Configuration" section for a description of how to set up your WLAN). If you've loaded the demo program onto your controller board as described in Appendix A and typed one of the functions **Ether_Web_Demo( )** or **WiFi_Web_Demo( )** to initialize the Wildcard and start the web demo, you can learn the EtherSmart or WiFi Wildcard's IP address by typing at the QED Terminal:

```
Ether_IP_Report( )
```

(Forth users would type the function names without any parentheses). The **Ether_IP_Report** function is described in the next section.

Assuming that the reported IP address is non-zero, you can then type the reported IP address into the web address bar of your web browser to see the demonstration web page. If the reported IP address is 0.0.0.0, then there is no DHCP service running on your LAN to assign an IP address, and you must explicitly assign one.

To explicitly assign an IP address and/or gateway IP and netmask, make sure that the Ethernet task is running by calling one of the functions **Ether_Setup_Default( )**, **Ether_Task_Setup_Default( )**, **Ether_Web_Demo( )**, **WiFi_Setup_Default( )**, **WiFi_Task_Setup_Default( )** or **WiFi_Web_Demo( )**. Then invoke the **Ether_Local_IP** and/or **Ether_Gateway** functions with the desired IP parameters, and then call **Ether_XPort_Update**. The presence of the non-zero IP address will override any DHCP value and disable DHCP on the EtherSmart/WiFi Wildcard. Likewise, specifying a non-zero gateway IP address or netmask will override any gateway IP and netmask values and disable DHCP.

The **Ether_Set_Local_IP** function in the demo program excerpt in Listing 1-4 makes it easy to interactively assign an IP address to the EtherSmart or WiFi Wildcard from the terminal window. For example, assuming that you've already invoked one of the initialization routines **Ether_Setup_Default( )**, **Ether_Task_Setup_Default( )**, **Ether_Web_Demo( )**, **WiFi_Setup_Default( )**, **WiFi_Task_Setup_Default( )** or **WiFi_Web_Demo( ),** you can set the IP address to 10.0.1.22 by typing the following into the QEDTerm terminal window:

```
DECIMAL
Ether_Set_Local_IP( 10, 0, 1, 22 )
```

Following the sample source code of **Ether_Set_Local_IP**, you can create a function to invoke the **Ether_Gateway** function to specify the gateway IP address and netmask. Then, to instantiate the values in Lantronix flash, execute **Ether_Update** followed by **Ether_Await_Response**.

To use an IP address that is automatically assigned by the LAN's gateway host, specify an IP address, gateway and netmask of 0.0.0.0 (the factory default). In this case, the Lantronix hardware relies on DHCP (Dynamic Host Configuration Protocol) running on the local area network's gateway server to set the IP address. To revert to the factory defaults which rely on a DHCP-assigned IP, gateway IP and netmask, execute **Ether_Xport_Defaults** or the convenient interactive version **Ether_Set_Defaults** as discussed above (see the demo program excerpt in Listing 1-4).

Reporting Routines

The Lantronix device on the EtherSmart/WiFi Wildcard implements a "monitor mode" than enables the examination of some of the parameters stored in the Lantronix flash. This mode also allows "ping" diagnostic packets to be sent to a specified IP address on the local area network. Unfortunately, these capabilities require a Lantronix reset sequence, and are slow, taking 7 seconds for a status report, and over 13 seconds for a ping report. Consequently, they are most useful for interactive diagnostics as opposed to runtime use. All of these functions require that you have initialized the Ethernet task using one of the functions **Ether_Setup_Default( )**, **Ether_Task_Setup_Default( )**, **Ether_Web_Demo( )**, **WiFi_Setup_Default( )**, **WiFi_Task_Setup_Default( )** or **WiFi_Web_Demo( )**.

The demo program excerpt in Listing 1-4 includes the convenient interactive **Ether_IP_Report** function which simply passes the **E_MODULENUM** constant to the **Ether_IP_Info_Report** routine. As described above, the **_Q** tag makes the function interactively callable from the terminal window. To use it, make sure that you have initialized the Wildcard, then type at your terminal

```
Ether_IP_Report(  )
```

with at least one space after the **(** character. If the info request was successful, the report is of the form:

```
IP 010.000.001.022 GW 010.000.001.002 Mask 255.255.255.000
```

This summarizes the IP address, Gateway (GW) IP address, and netmask. If there is a problem, an error message is displayed. Note that the Lantronix firmware may report indeterminate results if DHCP (Dynamic Host Configuration Protocol) is enabled but there is no active network connection.

The two IP information functions defined in the driver comprise a non-printing function called **Ether_IP_Info_Request**, and a printing version called **Ether_IP_Info_Report**. These work for both EtherSmart and WiFi Wildcards. **Ether_IP_Info_Request SEND**s a message via the **ether_command** mailbox to the task running the **Ether_Service_Loop** which dispatches the action function. This function enters the Lantronix monitor mode and retrieves the local IP address, gateway IP address, and netmask that are currently in use by the specified EtherSmart/WiFi Wildcard. After calling this routine the application must clear the **ether_response** mailbox using **Ether_Check_Response** or **Ether_Await_Response**, but note that the result will not be present until over 7 seconds have elapsed, so please be patient. This routine can be used to discover which IP address, gateway and netmask were automatically assigned by the DHCP (Dynamic Host Configuration Protocol) server on the Local Area Network (LAN). After the **ether_response** mailbox has been successfully read, the returned results are available by fetching 4 bytes each from the **Ether_My_IP_Ptr**, **Ether_Gateway_IP_Ptr**, and **Ether_Netmask_Ptr** locations. The counted ASCII response string from the Lantronix device is stored as a 2-byte count followed by the string data at **Ether_Outbuf**. **Ether_IP_Info_Report** builds on this function, automatically waiting for the response and printing the output string as described above.

Using "Ping" for Diagnostics

Ping is a computer network tool used to test whether a particular host is reachable across an IP network. Ping works by using the ICMP (Internet Control Message Protocol) to send "echo request" packets to the target host and listening for ICMP "echo response" replies. Using interval timing and response rate, ping estimates the round-trip time between hosts. Ping was named after active sonar in submarines, in which a pulse of energy (analogous to the network packet) is aimed at the target, which then bounces from the target and is received by the originator.

The demo program excerpt in Listing 1-5 includes an interactive function named **Ether_Ping** that pings a remote computer at a specified IP address. As usual, this function requires that you have initialized the Ethernet task using one of the functions **Ether_Setup_Default( )**, **Ether_Task_Setup_Default( )**, **Ether_Web_Demo( )**, **WiFi_Setup_Default( )**, **WiFi_Task_Setup_Default( )** or **WiFi_Web_Demo( )**. For example, to ping a computer at the remote IP address 10.0.1.3, simply type the following into the terminal window:

```
Ether_Ping( 10, 0, 1, 3)
```

If there is a problem, the appropriate error message "Couldn't enter monitor mode" or "No response from remote" is printed. If there is a ping response, a report of the following form is printed:

```
Seq 001 time 10ms
Seq 002 time 10ms
Seq 003 time 10ms
Seq 004 time 10ms
Seq 005 time 10ms
Seq 006 time 10ms
```

The printed display specifies the round-trip travel times between the EtherSmart/WiFi Wildcard and the specified remote host.

Listing 1-5 Interactive Ping Function from the Demo Program.

```
_Q void Ether_Ping(int remote_ip1, int remote_ip2, int remote_ip3,  int remote_ip4)
// call this AFTER calling main() or Ether_Web_Demo or Ether_Task_Setup_Default()
// or WiFi_Web_Demo or WiFi_Task_Setup_Default.
// on error, prints " Couldn't enter monitor mode!" or " No response from remote".
// on error, prints " Couldn't enter monitor mode!" or " No response from remote".
// takes thirteen seconds to execute, so be patient.
// Report is of the form (summarizes response time from specified remote host):
// Seq 001 time 10ms
// Seq 002 time 10ms
// Seq 003 time 10ms
// Seq 004 time 10ms
// Seq 005 time 10ms
// Seq 006 time 10ms
// NOTES: type DECIMAL at the monitor before invoking this function interactively!
//        The input types are declared as int to simplify interactive calling,
//        as the interactive debugger would require char specifiers before each input
//        parameter if the char type were used.
{  Ether_Ping_Report(remote_ip1, remote_ip2, remote_ip3, remote_ip4, E_MODULENUM);
}
```

The two ping functions defined in the EtherSmart/WiFi driver comprise a non-printing function called **`Ether_Ping_Request`**, and a printing version called **`Ether_Ping_Report`**. These behave in ways that are parallel to **`Ether_IP_Info_Request`** and **`Ether_IP_Info_Report`** as described in the prior section.

`Ether_Ping_Request` **`SEND`**s a message via the **`ether_command`** mailbox to the task running the **`Ether_Service_Loop`** which dispatches the action function. After calling this routine the application must clear the **`ether_response`** mailbox using **`Ether_Check_Response`** or **`Ether_Await_Response`**, but note that the result will not be present until over 13 seconds have elapsed, so please be patient. After the **`ether_response`** mailbox has been successfully read, the counted ASCII response string is stored as a 2-byte count followed by the string data at **`Ether_Outbuf`**. **`Ether_Ping_Report`** builds on this function, automatically waiting for the response and printing the output string as described above.

Shutting Down the Lantronix Device To Save Power

The EtherSmart and WiFi Wildcards draw power from the +5Volt regulated supply on the Wildcard bus. The Lantronix XPort draws 250 mA from a linear +3.3V regulator on the Wildcard. Shutting down the +3.3V regulator saves 1.25 Watts. The WiPort draws about 300 mA of current. The regulator can be shut down under program control while the WiPort is not in use to save 1.5 Watts of power. This can extend operational time in battery powered systems. To attain these power savings, call the **`Ether_Shutdown`** function. Of course, the Ethernet/WiFi interface on the specified wildcard is not useable while in the shutdown state. To revert to the default powered-up state, power cycle the hardware, or execute **`Ether_Init, WiFi_Init`** or one of their calling functions listed in Table 1-3.

Initializing Multiple EtherSmart and/or WiFi Wildcards

The standard driver and demo software enables you to install two EtherSmart and/or WiFi Wildcards on a single controller, using one for code downloads and interactive communications ("revectored serial via TCP/IP") and the other for communications services such as email, serial tunneling, and web service. Simply store the modulenum of the revectored serial Wildcard into the **`ether_revector_module`** variable as illustrated by the **`Ether_Monitor_Demo`** function in the demonstration program as described below. **`Ether_Monitor_Demo`** works for EtherSmart and WiFi Wildcards. Initialize the other Wildcard for communications services using the standard initialization functions such as **`Ether_Setup_Default( )`**, **`Ether_Task_Setup_Default( )`**, **`Ether_Web_Demo( )`**, **`WiFi_Setup_Default( )`**, **`WiFi_Task_Setup_Default( )`** or **`WiFi_Web_Demo( )`**.

If you are using more than one EtherSmart or WiFi Wildcard to provide communications services (serial tunneling, email, and web service), you will need to define a custom version of **`Ether_Service_Loop`**, and a custom version of **`Ether_Task_Setup`** or **`WiFi_Task_Setup`** for each additional Wildcard. Each Wildcard that runs communications services must have its own task running the service loop, with its own set of buffers and mailboxes.

`Ether_Service_Loop` is a simple infinite loop function; it works for both EtherSmart and WiFi Wildcards. In the loop body are calls to **`Ether_Connection_Manager`**, **`Ether_Command_Manager`**, and **`Pause`**. The default version of **`Ether_Service_Loop`** specifies the contents of the **`ether_service_module`** variable as the parameter passed to **`Ether_Connection_Manager`** and **`Ether_Command_Manager`** in the infinite loop body. Your custom version should not use this variable; rather, it should specify the modulenum of the additional EtherSmart/WiFi Wildcard that you are adding.

If you are adding an EtherSmart Wildcard, your custom version of **`Ether_Task_Setup`** should call **`Ether_Setup`** with a unique buffer base xaddress, mailbox base address, and the modulenum of the additional EtherSmart Wildcard. It must then build a new task and call **`ACTIVATE`** to install the custom version of the Ethernet service loop function as described in the prior paragraph. If you are adding a WiFiSmart Wildcard, your custom version of **`WiFi_Task_Setup`** should call **`WiFi_Setup`** with a unique buffer base xaddress, mailbox base address, and the modulenum of the additional WiFi Wildcard. It must then build a new task and call **`ACTIVATE`** to install the custom version of the Ethernet service loop function

Using this technique, multiple EtherSmart and/or Wildcards can be configured to provide communications services. If you have questions, feel free to contact Mosaic Industries for assistance.

Enabling the Encryption Feature

The Lantronix XPort and WiPort devices support 256 bit AES Rijndael encryption. Any AES-enabled devices that share the same key can communicate securely with one another. This encryption is not to be confused with WiFi security encryption; the AES encryption is separate and independent from WiFi security.

Rijndael is the block cipher algorithm chosen by the National Institute of Science and Technology (NIST) as the Advanced Encryption Standard (AES) to be used by the US government. Configuring two or more Lantronix devices with the same keys and key length allows them to communicate with one another on a network, while preventing anyone who does not know the encryption key from deciphering the network data.

To enable AES encryption, initialize the EtherSmart/WiF Wildcard using one of the initialization functions in Table 1-3. Store a 32-byte, 256 bit key into a buffer in memory and then invoke **`Ether_Encryption`**, passing it the base xaddress and count of the key buffer, plus the relevant modulenum. Make sure there are no active connections during the execution of this function, as they will be interfered with when monitor mode is entered. Double check the key values, as there is no way to read them back after they are set. The function requires approximately 13 seconds to complete.

An alternate way to configure the encryption settings is to go into "setup mode" by connecting to port 9999 at the Lantronix device's known IP address using the "raw" data transfer mode of a free network terminal program such as Putty. (See the section titled "Using the Free Putty TCP/IP Terminal for Testing and Development"). Once connected to the Lantronix device using Putty, hit the "Enter" key within 3 seconds. Then choose option 6 (security), then follow the prompts to enable encryption, choose the key length, and change or enter a key.

To return to non-encrypted operation, make sure that the Ethernet task is running by executing **`Ether_Task_Setup_Default`** or **`WiFi_Task_Setup_Default`** or another appropriate initialization function. Then execute **`Ether_XPort_Defaults`**; this works for both EtherSmart and WiFi Wildcards.

Lantronix asks that we include this product notice: This and other devices that implement encryption cannot be exported or re-exported to a national resident of Cuba, Iran, North Korea, Sudan, Syria or any other country to which the United States has embargoed goods; see the Lantronix documentation and website for details.

WiFi Wildcard Configuration

The WiFi Wildcard ships in a default configuration that specifies the "infrastructure" (access point) mode with reasonable values for transmit power level and rate. All WiFi security is disabled by default; this makes it simpler to verify proper WiFi operation when first installing the Wildcard in a network. The default case-sensitive SSID (Service Set ID) is

WIFI_WILDCARD

The SSID and the security settings must be shared by all nodes on the WLAN in order for mutual communications to be established.

There are two ways to modify the WiFi configuration and security settings:

### by calling the functions listed in Table 1-5; or,

### by using the Lantronix internal configuration webserver available on port 8000 at the Wildcard's IP address.

Each of these methods is discussed in turn.

Table 1-5 WiFi configuration functions and constants.

`WIFI_CCMP_GROUP_ENCRYPT`	`WIFI_TKIP_GROUP_ENCRYPT`
`WIFI_CCMP_PAIR_ENCRYPT`	`WIFI_TKIP_PAIR_ENCRYPT`
`WiFi_Check`	`WIFI_WEP128_PAIR_ENCRYPT`
`WiFi_Encryption_Key`	`WIFI_WEP64_PAIR_ENCRYPT`
`WIFI_NO_SECURITY`	`WIFI_WEP_GROUP_ENCRYPT`
`WiFi_Options`	`WIFI_WEP_SUITE`
`WiFi_Security`	`WIFI_WPA2_SUITE`
`WiFi_SSID`	`WIFI_WPA_SUITE`

The WiFi_SSID Function

`WiFi_SSID` accepts a buffer specification (xaddress and count) and a modulenum. For the specified WiFi Wildcard module, it writes a reference to the specified WiFi SSID ("Service Set ID") buffer into the **`ether_info`** structure. Assuming that you have initialized the Ethernet task by calling **`WiFi_Task_Setup`**, you can instantiate the string into the Lantronix WiPort flash after invoking this function by executing **`Ether_XPort_Update`** followed by **`Ether_Await_Response`**. The case-sensitive SSID is the name which must be shared among members of a WLAN (Wireless Local Area Network) in order for the members to associate with one another. The default SSID set by Mosaic at the factory, and restored by **`Ether_XPort_Defaults`**, is "WIFI_WILDCARD" (without the quotes).

The WiFi_Options Function

The **`WiFi_Options`** function configures the transmit power level and rate, and specifies infrastructure versus ad-hoc mode. The function writes the specified parameters into the **`ether_info`** struct for the specified WiFi module. These parameters do not take effect until **`Ether_Xport_Update`** is invoked as explained below. See the glossary entry of **`WiFi_Options`** for a detailed list of the parameters passed to this function and their meanings. Briefly, the transmit power can be set to 0, 6, 12, or 18 dBm. dBm is a measure of power in dB relative to 1 milliWatt of radiated power: 0 dBm is equivalent to 1 milliWatt, and each additional 3 dBm represents roughly a doubling of power. A power management flag enables transmitter power management if nonzero, and disables it if zero (the default). The transmit data rate can be set to one of 8 values in the range from 1 Mbps to 54 Mbps, with a default of 18 Mbps (Megabits per second). A flag enables automated control of transmission rate if nonzero (the default), and disables automated rate control if zero. An additional flag specifies the "ad hoc" mode if nonzero. In ad hoc mode, two wireless devices communicate directly with each other without the use of an access point. A zero adhoc_flag (the default) configures the Wildcard for "infrastructure mode" which is the standard Wireless Local Area Network configuration that uses an access point to enable communications among multiple wireless devices. If the adhoc_flag is true, then the ad hoc channel number can be set to a specified value. In the United States and Canada, allowed ad hoc channels are between 1 and 11 inclusive.

After executing the **WiFi_Options** function, assuming that you have initialized the Ethernet task by calling **WiFi_Task_Setup**, you can instantiate the values into the WiPort flash by executing **Ether_XPort_Update** followed by **Ether_Await_Response**. To restore the values marked as defaults, execute **Ether_XPort_Defaults** followed by **Ether_Await_Response**.

The WiFi_Security Function

WiFi encryption prevents unauthorized parties from eavesdropping on communications that occur on the secured WLAN. The most common security methods are called WEP and WPA. WEP, or Wired Equivalent Privacy, is the oldest and simplest WiFi security suite. WEP64 uses a 40-bit encryption key with a 24-bit initialization vector, and WEP128 uses a 104-bit encryption key with a 24-bit initialization vector. WEP is not completely immune to attack, and the newer WPA (WiFi Protected Access) offers stronger encryption. TKIP (Temporal Key Integrity Protocol) pairwise encryption method is used in conjunction with WPA, and uses a 128 bit key. The still newer WPA2 is associated with the 802.11i protocol, and uses the CCMP pairwise encryption method.

Note that WiFi encryption is independent of and not related to the AES encryption described in the **Ether_Encryption** glossary entry.

The **WiFi_Security** function configures the security parameters for the WiFi Wildcard. Its C function prototype is as follows:

```
void WiFi_Security (int authenticate_flag, int suite_method,
                   int pairwise_encrypt_method, int group_encrypt_method,
                   int key_length, int passphrase_flag, int modulenum )
```

This function writes the specified parameters into the **ether_info** struct for the specified WiFi module. These parameters do not take effect until **Ether_Xport_Update** is invoked. The default upon shipment from Mosaic or after executing **Ether_XPort_Defaults** is “no security”, equivalent to specifying zeros for all of the security parameters passed to the function.

A nonzero **authenticate_flag** parameter enables authentication, while a zero disables it. The **suite_method** parameter should be one of the constants **WIFI_NO_SECURITY**, **WIFI_WEP_SUITE**, **WIFI_WPA_SUITE**, or **WIFI_WPA2_SUITE**. WEP, or Wired Equivalent Privacy, is the oldest and simplest WiFi security suite. It is not completely immune to attack, and the newer WPA (WiFi Protected Access) offers stronger encryption. The still newer WPA2 is associated with the 802.11i protocol. The **pairwise_encrypt_method** parameter should be one of the constants **WIFI_NO_SECURITY**, **WIFI_WEP64_PAIR_ENCRYPT**, **WIFI_WEP128_PAIR_ENCRYPT**, **WIFI_TKIP_PAIR_ENCRYPT**, or **WIFI_CCMP_PAIR_ENCRYPT**. The pairwise method specifies the main encryption scheme for the security suite. WEP64 uses a 40-bit encryption key with a 24-bit initialization vector, and WEP128 uses a 104-bit encryption key with a 24-bit initialization vector. TKIP (Temporal Key Integrity Protocol) pairwise encryption method is used in conjunction with WPA, and uses a 128 bit key. CCMP pairwise encryption is used in conjunction with WPA2 on 802.11i networks.

The `group_encrypt_method` should be one of the constants `WIFI_NO_SECURITY`, `WIFI_WEP_GROUP_ENCRYPT`, `WIFI_TKIP_GROUP_ENCRYPT`, or `WIFI_CCMP_GROUP_ENCRYPT`. Group encryption is typically not used if the WEP security suite is specified. The standard group encryption for the WPA suite is TKIP, although the group encryption should be set to `WIFI_WEP_GROUP_ENCRYPT` if the WPA suite is specified with encryption called “TKIP plus WEP group keys”.

The `key_length` parameter can be safely set to zero, or it can be set to the actual key length. Testing indicates that the current Lantronix firmware (V6.1.0.1) ignores the key length field and uses the encryption type to infer the length of a hex key, and uses a trailing zero to determine the length of a passphrase key.

The `passphrase_flag` parameter should be nonzero if an 8- to 63-byte passphrase is used to specify the encryption key, and should be zero if a hex key is specified. The key type should match the parameters passed to the `WiFi_Encryption_Key` function explained in the next section. Both WEP and WPA keys can be specified using hex digits, or using a passphrase of 8 to 63 bytes that is converted into a key using a hash function. See the next section for a function that sets the security key.

After executing the `WiFi_Security` function, assuming that you have initialized the Ethernet task by invoking `WiFi_Task_Setup`, you can instantiate the values into the WiPort flash by executing `Ether_XPort_Update` followed by `Ether_Await_Response`. To restore the default “no security” values, execute `Ether_XPort_Defaults` followed by `Ether_Await_Response`.

The WiFi_Encryption_Key Function

`WiFi_Encryption_Key` accepts a buffer xaddress and count containing an encryption key (hex or passphrase), and installs the specified WiFi encryption key into the WiPort flash. The number of key bytes stored is clamped to a maximum of 63 bytes, which is the limit for a passphrase key as described below. If the number of key bytes is less than 63, the unspecified bytes are set to zero. The most significant byte of the key is the byte stored at the beginning of the buffer. All devices on a WiFi Local Area Network must have the same SSID, WiFi security settings, and key to be able to communicate with one another. Devices that do not know the security key will in theory not be able to eavesdrop on the secure communications.

There are two ways of specifying a key: as a hexadecimal (hex) sequence, or as a passphrase. Specifying a hex key is simple: just type in a hexadecimal number of the correct length. Each hex character (in the range 0 to 9, A to F) specifies 4 bits of the key. A passphrase is 8 to 63 printable ASCII bytes that is automatically converted into 1 to 4 numeric keys by a built-in hash function. If you are offered a choice in a configuration screen, always select the first key (key index 0) as this is the one the Lantronix device uses. The `WiFi_Encryption_Key` routine ensures that a trailing zero (null delimiter) is stored after the passphrase, as required by the Lantronix firmware. A passphrase can be used with WEP or WPA.

A WEP64 hex key is 40 bits, corresponding to ten hex digits. An example 40-bit key is:

```
123456789A
```

A WEP128 hex key is 104 bits, corresponding to 26 hex digits. An example 104-bit key is:

```
123456789ABCDEF0123456789A
```

A WPA/TKIP key is 128 bits, corresponding to 32 hex digits, An example 128-bit key is:

```
123456789ABCDEF0123456789ABCDEF0
```

After a restart, initialize the WiFi Wildcard using **`WiFi_Setup`** or one of its calling functions listed in Table 1-3, and then invoke **`WiFi_Encryption_Key`**, passing it the base xaddress and size of the key that has been stored in memory. Make sure there are no active connections during the execution of this function, as they will be interfered with when monitor mode is entered. Double check the key values, as there is no way to read them back after they are set. At this point the key is present, but WiFi encryption is not enabled until you setup WiPort flash record 8 via the functions **`Wifi_Security`**, **`WiFi_SSID`**, and **`WiFi_Options`**, and then execute **`Ether_XPort_Update`** as described in the preceding sections. To undo the effect of these commands and return to non-encrypted operation of the WiFi Wildcard, make sure that the Ethernet task is running by invoking **`WiFi_Task_Setup`**. Then execute **`Ether_XPort_Defaults`** (see its glossary entry; it works for both EtherSmart and WiFi Wildcards). Remember to adjust the security settings on your wireless access point, as all security parameters must match for communications to occur.

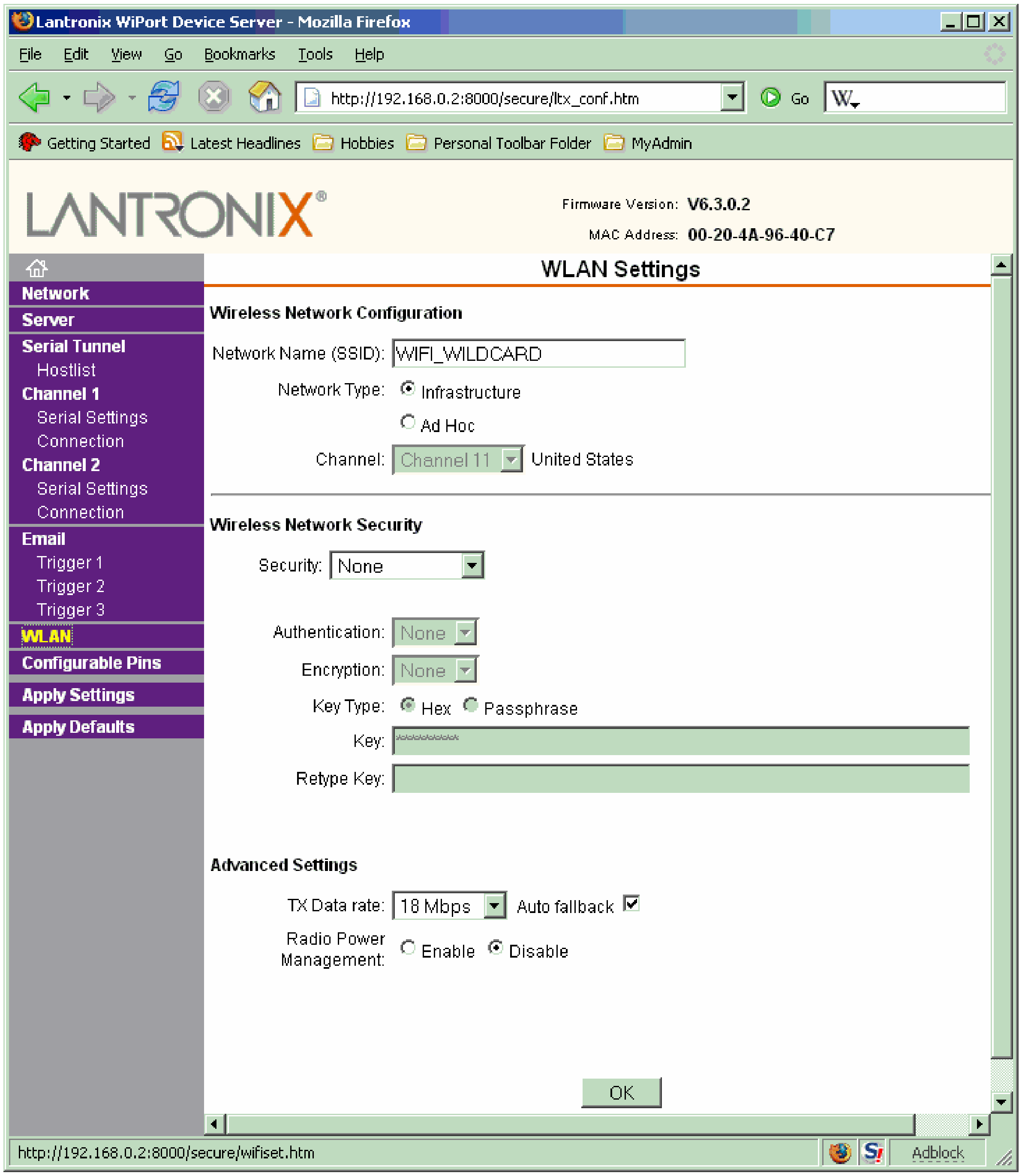
Using the Lantronix WiPort Configuration Webserver To Configure Security

An alternative method to customize the wireless and security settings of the WiFi Wildcard is to log into the configuration web site served out by the Lantronix WiPort. Assuming that you have established wireless communications (see the section titled "Verifying WiFi Wildcard Communications"), and you know its IP address, you can use your PC's browser to access its internal configuration website on port 8000. For the WiFi Wildcard with IP address 192.168.0.2, you would type the following into the address bar of your web browser:

http://192.168.0.2:8000

The :8000 tells the browser to go to port 8000 instead of the standard web port 80. The WiFi Wildcard uses port 80 for all of its standard network traffic, and locates the built-in configuration webserver at port 8000.

When the login box comes up, just hit "Enter" (no password required), and you should see the Lantronix configuration web page. The column at the left allows you to select which settings are to be configured. Click on the "WLAN" item to bring up the screen shown in Figure 1-9.



Firmware Version: V6.3.0.2
MAC Address: 00-20-4A-96-40-C7

WLAN Settings

Wireless Network Configuration

Network Name (SSID): WIFI_WILDCARD
Network Type: [✓ Infrastructure] Ad Hoc
Channel: Channel 11 United States

Wireless Network Security

Security: None
Authentication: None
Encryption: None
Key Type: [✓ Hex] Passphrase
Key: **********
Retype Key:

Advanced Settings

TX Data rate: 18 Mbps Auto fallback ✓
Radio Power Management: Enable [✓ Disable]

http://192.168.0.2:8000/secure/wifiset.htm

Figure 1-9 Default WLAN settings page served out by the WiPort's internal configuration webserver on port 8000.

<u>A good rule is to use the WiPort configuration webserver to change ONLY the "WLAN" settings which include the SSID and the wireless security settings.</u> Changing other settings could render the WiFi Wildcard inoperable. (It is permissible to use the "Network" screen if you want to hard code an IP address and subnet; these can also be configured as described in the section above titled "Configuring the Lantronix Device").

If by any chance any settings are corrupted, use the **`Ether_XPort_Defaults`** function described in this document to restore the Mosaic factory settings. Do not use the “Apply Factory Defaults” button in the configuration website. The “Apply Factory Defaults” button does NOT establish the correct Mosaic defaults; Mosaic’s defaults are different than those selected by Lantronix.

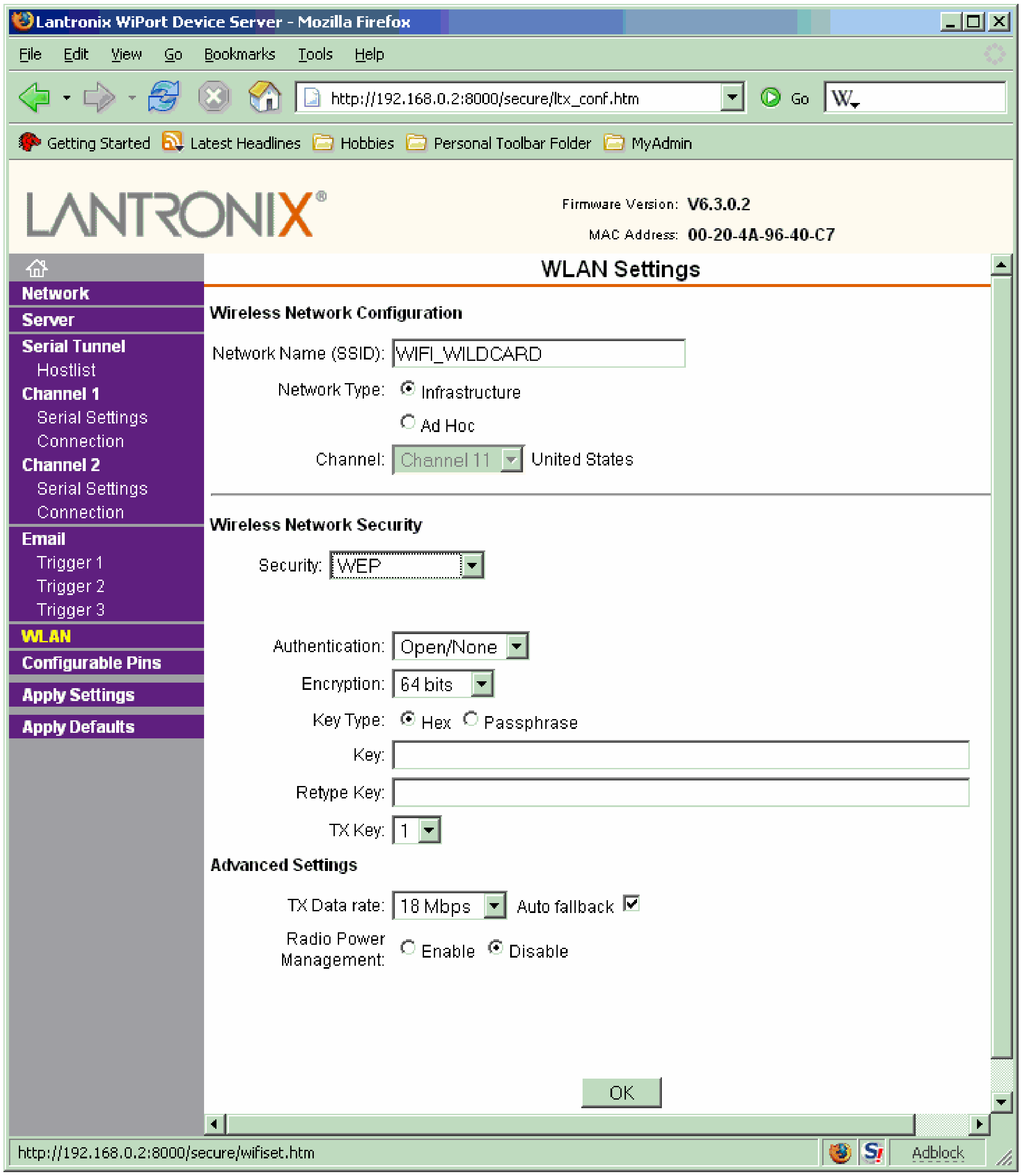
Figure 1-9 shows the WLAN configuration page of the Lantronix configuration site in its default state with security disabled and the SSID equal to “WIFI_WILDCARD”. You can use this page to change the SSID, specify ad hoc (as opposed to infrastructure) mode, enable WiFi security, specify the encryption key, and change the transmit data rate and power settings. To instantiate the changed fields, click the “OK” button at the bottom of the screen, and then click “Apply Settings” from the left-hand menu. The Lantronix device will install the new settings in its flash memory and reboot; this takes over 10 seconds. Of course, once you change the SSID or enable security on any part of the WLAN, the same SSID, security key and security type must be setup on the other members of the WLAN to enable communications.

We examine each of the fields shown in Figure 1-9 in turn. The “Network name (SSID)” contains the default case-sensitive WIFI_WILDCARD Service Set ID. You can type any valid ASCII string up to 32 bytes long into this field. Make sure to configure the same SSID on any other devices on your WLAN, as matching SSID’s are required for wireless association to occur.

The default “Network Type” is infrastructure, meaning a network that incorporates a wireless access point to coordinate communications among wireless nodes. The alternative is “ad hoc” mode, which is a point-to-point network between two wireless devices. For example, you could configure two WiFi Wildcards in ad hoc mode and set them up to communicate with one another. If the ad hoc mode is specified, the “Channel” box is enabled, allowing you to specify a WiFi channel number between 1 and 13, inclusive. Channels 1 through 11 are valid in the U.S. and Canada. Channel 11 is the default.

The “Security” field contains the default value “None”; when this option is selected, the remaining security fields are “greyed out” and cannot be modified. The alternative entries in the “Security” field are “WEP”, “WPA”, and “802.11i/WPA2”. These are explained in detail in the section titled “The WiFi_Security Function”. If you specify WEP, WPA, or WPA2 encryption, the remaining security fields are enabled.

Figure 1-10 shows the WLAN settings page when the WEP security suite is selected. The default “Authentication” for WEP is “open/none”. The alternative is “shared”. This field corresponds to the “authentication_flag” parameter passed to the **`WiFi_Security`** function described above, where “open/none” corresponds to a zero flag, and “shared” corresponds to a nonzero authentication flag.


Lantronix WiPort Device Server - Mozilla Firefox

http://192.168.0.2:8000/secure/ltx_conf.htm

Firmware Version: V6.3.0.2
MAC Address: 00-20-4A-96-40-C7

WLAN Settings

Wireless Network Configuration

Network Name (SSID): WIFI_WILDCARD
Network Type: [✓ Infrastructure] Ad Hoc
Channel: Channel 11 United States

Wireless Network Security

Security: WEP

Authentication: Open/None
Encryption: 64 bits
Key Type: [✓ Hex] Passphrase
Key:
Retype Key:
TX Key: 1

Advanced Settings

TX Data rate: 18 Mbps Auto fallback ✓
Radio Power Management: Enable [✓ Disable]

http://192.168.0.2:8000/secure/wifiset.htm

Figure 1-10 WLAN settings page with WEP security selected.

The next "Encryption" field allows the selection of 64-bit or 128-bit WEP encryption. These respectively correspond to the `WIFI_WEP64_PAIR_ENCRYPT` and `WIFI_WEP128_PAIR_ENCRYPT` "pairwise_encryption" flag passed to the `WiFi_Security` function described above.

The "Key Type" field can be specified as either "hex" or "passphrase". The WEP encryption key can be entered as a hexadecimal value, or as a "passphrase". A passphrase is an 8 to 63 byte printable ASCII string that is processed by a "hash function" to create one or more numeric keys. If a passphrase is used, the WiFi Wildcard always uses key index 0 generated by the passphrase hash function. The configuration page asks for the key to be typed twice to help ensure that there is no entry error; keys cannot be read back from the Lantronix device once they are installed.

If "hex" key type is selected, note that a WEP64 hex key is 40 bits, corresponding to ten hex digits. An example 40-bit WEP key is:

```
123456789A
```

A WEP128 hex key is 104 bits, corresponding to 26 hex digits. An example 104-bit key is:

```
123456789ABCDEF0123456789A
```

Figure 1-10 shows the WLAN settings page when the WPA security suite is selected. The default (and only) "Authentication" option for WPA is "Pre-Shared Keys (WPA-PSK)". This field corresponds to the "authentication_flag" parameter passed to the **WiFi_Security** function described above, where a true (nonzero) authentication flag must always be passed if the WPA security suite is in use.

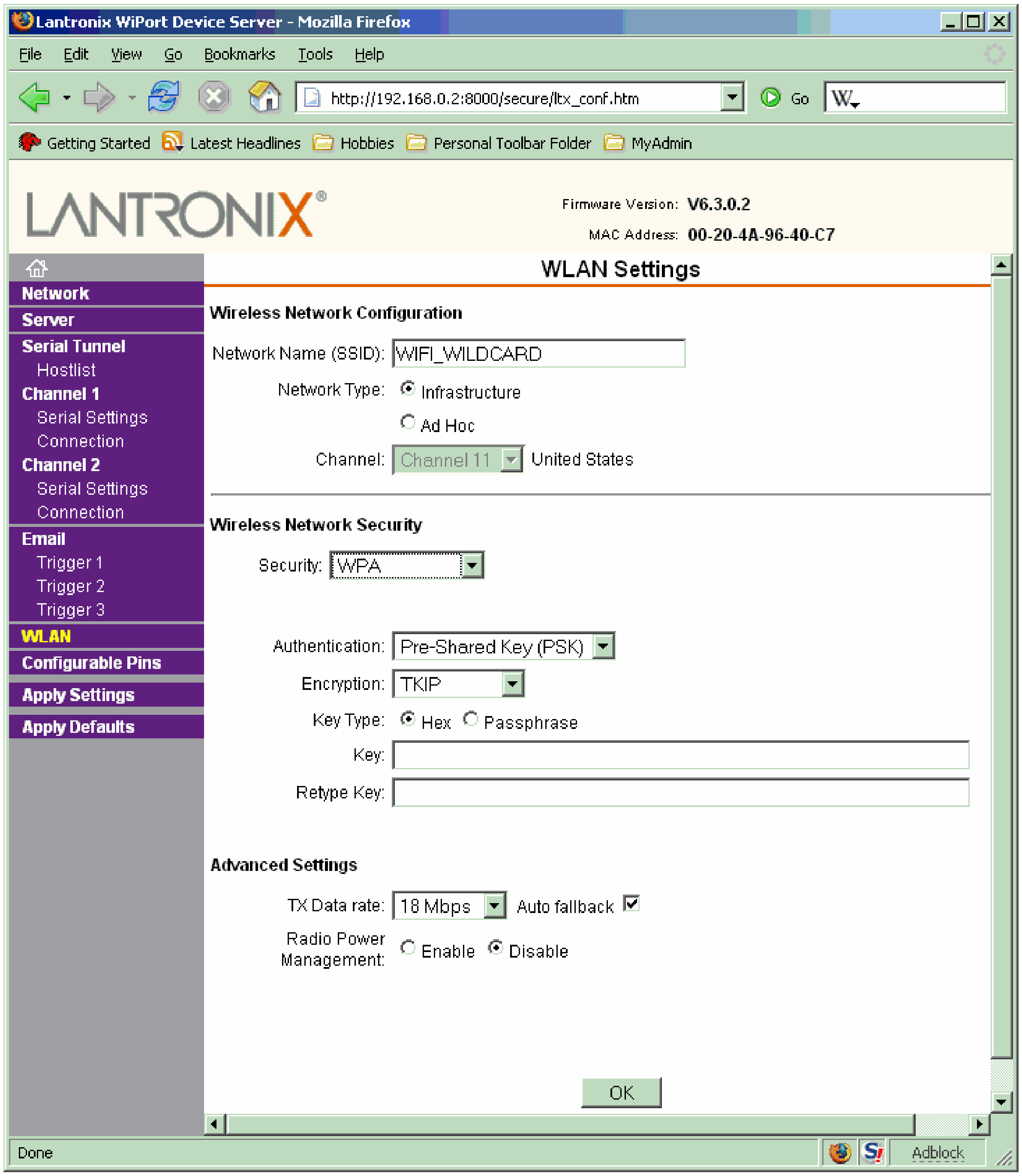
The next "Encryption" field allows the selection of "TKIP" or "TKIP+WEP group keys" encryption. In each case, the pairwise encryption method is TKIP (Temporal Key Integrity Protocol) pairwise encryption method which is used in conjunction with WPA. This corresponds to the **WIFI_TKIP_PAIR_ENCRYPT** "pairwise_encryption" flag passed to the **WiFi_Security** function described above. If WPA is in use, the encryption method also specifies a "group encryption" method. Choosing "TKIP" encryption is equivalent to specifying **WIFI_TKIP_GROUP_ENCRYPT** as the "group_encryption" flag passed to the **WiFi_Security** function described above. Choosing "TKIP+WEP group keys" encryption with the WPA security suite is equivalent to specifying **WIFI_WEP_GROUP_ENCRYPT** as the "group_encryption" flag passed to the **WiFi_Security** function described above.

The "Key Type" field can be specified as either "hex" or "passphrase". The passphrase can be 8 to 32 ASCII bytes. If the "hex" key type is selected, note that a WPA/TKIP key is 128 bits, corresponding to 32 hex digits, An example 128-bit WPA key is:

```
123456789ABCDEF0123456789ABCDEF0
```

The remaining settings in Figure 1-10 specify the data rate (with or without automated rate control), and whether automated radio power management is enabled.

To confirm any changes you have made in the WLAN configuration page, click "OK" at the bottom of the page, and then click "Apply Settings" to store the changes into flash in the Lantronix device. As stated earlier, do NOT click the "Apply Factory Defaults" button, as this will corrupt the WiFi Wildcard communications with the Mosaic controller. To recover from this or any other configuration problem and return to the Mosaic defaults, execute **Ether_Xport_Defaults** or the convenient interactive version **Ether_Set_Defaults** defined in the demonstration code (see the demo program excerpt in Listing 1-4).



Firmware Version: V6.3.0.2
MAC Address: 00-20-4A-96-40-C7

WLAN Settings

Wireless Network Configuration

Network Name (SSID): WIFI_WILDCARD
Network Type: [✓ Infrastructure] Ad Hoc
Channel: Channel 11 United States

Wireless Network Security

Security: WPA

Authentication: Pre-Shared Key (PSK)
Encryption: TKIP
Key Type: [✓ Hex] Passphrase
Key:
Retype Key:

Advanced Settings

TX Data rate: 18 Mbps Auto fallback ✓
Radio Power Management: Enable [✓ Disable]

Figure 1-11 WLAN settings page with WPA security selected..

Code Downloads and Interactive Communications via TCP/IP

The EtherSmart/WiFi Wildcard can be configured to replace the serial port on the controller, enabling code downloads and interactive development to be performed via Ethernet or WiFi instead of via RS-232. This is called "revectored serial via TCP/IP" and is conceptually very simple: the three fundamental serial primitives **Key**, **AskKey**, and **Emit** are "revectored" so that they use the Ethernet or WiFi port on the Wildcard instead of the serial port on the controller.

The simplest approach to revectoring is to store the correct modulenum of the Wildcard into the global variable named **ether_revector_module**, and then to call **Ether_Serial_Revector** to revector the serial primitives. This runs the default (startup) task through the specified EtherSmart/WiFi Wildcard with no additional data structures needed. In other words, once this action is performed, you no longer communicate with the controller's operating system monitor via the standard RS-232 terminal; all communications must take place via an Ethernet or WiFi connection using a TCP/IP terminal such as Putty.

For information about how to download and use the Putty terminal, see the section above titled "Using the Free Putty TCP/IP Terminal for Testing and Development". Briefly, after a simple Google search for "Putty", download the small executable, and double click its icon to start it. Simply type the Wildcard's IP address (such as 10.0.1.22) and port 80 into the boxes in the Putty configuration screen, select the "Rlogin" mode, and click the "Open" button to establish the connection. C programmers may notice a 1-line response from the Mosaic controller in the form:

```
XTERM/38400SSP ?
```

This message can be safely ignored; it results from an unrecognized login message from the Rlogin protocol that passes through the Lantronix device to the operating system.

Ether_Serial_Revector revectors the serial primitives of the current task (that is, the task that invokes this routine) to use the EtherSmart or WiFi Wildcard that is specified by the contents of the **ether_revector_module** variable. Make sure to explicitly store a value to the **ether_revector_module** variable before invoking **Ether_Serial_Revector**, as the variable is not initialized by default. **Ether_Serial_Revector** stores the xcfa (32-bit execution address) of **E_Emit** into the user variable **UEMIT**, stores the xcfa of **E_Ask_Key** into **UASK_KEY** (**U?KEY** in Forth), and stores the xcfa of **E_ASCII_Ke**y into **UKEY**.

After invoking **Ether_Serial_Revector**, use the Putty "Rlogin" mode to connect to the local port (typically port 80) at the specified local IP address, and you're talking to the QED monitor via TCP/IP over Ethernet or WiFi. To revert to standard serial operation via QED Term, type **COLD** in the Putty terminal window to revert to standard serial, then from QEDTerm type **RESTORE** (or **RESTORE.ALL** on a V6.xx kernel machine) to bring back access to all compiled routines, and continue communications using QEDTerm.

If you want to maintain serial communications via QEDTerm with the default task while running a separate task with I/O revectored via the EtherSmart/WiFi Wildcard, then build and activate a task using **Ether_Monitor** as the activation routine. Use Putty "Rlogin" mode to connect to the local port (typically = 80) at the specified local IP address, and you're talking to the task via TCP/IP while simultaneously maintaining serial communications with the primary task. An example of this approach is presented in Listing 1-6 excerpted from the Demo program. The interactively callable **Ether_Monitor_Demo** routine from the demonstration program sets the **ether_revector_module** variable, and builds and activates a task to run the **Ether_Monitor** function. **Ether_Monitor** is an infinite task loop that calls **Ether_Serial_Revector** and then invokes the QED-Forth monitor routine named **QUIT**. Executing this routine, or using it as the activation function for a task causes serial I/O for the affected task to be revectored to the EtherSmart Wildcard whose modulenum is stored in the **ether_revector_module** global variable.

Listing 1-6 Demo Program Code To Implement Revectored Serial via TCP/IP.

```
// ***************** REVECTORED SERIAL VIA TCP/IP ************

TASK ether_montask;  // 1 Kbyte per task area

_Q void Ether_Monitor_Demo( int modulenum )
// works for xport or wiport (ethernet/wifi)
// builds and activates a monitor task for the specified module.
// this function expects the modulenum as an input parameter so that
// you can run 2 installed Ethersmart wildcards at once:
// one wildcard can run the standard services (web, email, tunneling)
// while the other one (running Ether_Monitor_Demo) is used to download code
// and control program development and debugging.
// To do this, first call main (which runs services on the E_MODULENUM wildcard),
// then call this Ether_Monitor_Demo function with a different modulenum.
{  ether_revector_module = modulenum;  // set to specify which module's revectored
   SERIAL_ACCESS = RELEASE_ALWAYS; // allow sharing of serial ports
 // NEXT_TASK = TASKBASE;// empties task loop; comment out if other tasks are allowed
   BUILD_C_TASK(0, 0, &ether_montask );
   ACTIVATE(Ether_Monitor, &ether_montask );
}
```

Serial Tunneling

"Serial tunneling" is a name for an exchange of serial data between two computers, typically using the TCP/IP protocol. In network terminology, a "tunnel" is a path that passes data without the sender and receiver having to know or care about the details of the transfer mechanism. "Serial tunneling" was originally coined as a term to describe a method for allowing instruments and computers that were designed to communicate via RS-232 to now communicate via TCP/IP. Its meaning has broadened to encompass a variety of serial data exchanges between instruments via Ethernet or WiFi. The EtherSmart/WiFi Wildcard can implement "Serial Tunneling" by initiating or accepting TCP/IP Ethernet connections and exchanging binary and/or ASCII text data with other devices on the local network.

A comprehensive suite of pre-coded driver functions is available to simplify the implementation of serial tunneling in your application program. These can be classified as buffer management, data transmission and reception, connection control, and inter-task service management functions. Each of these is discussed in turn in the context of the **Tunnel_Test** function from the demo program which works for both EtherSmart and WiFi Wildcards.

A Serial Tunneling Example

Listing 1-7 presents the interactively callable **Tunnel_Test** function from the demo program. Let's take a look at how this function works. If you have not already done so, initialize the

Wildcard by invoking an initialization function. If you have an EtherSmart Wildcard, you can type one of the following at the terminal prompt:

```
Ether_Task_Setup_Default(  )
```

or:

```
Ether_Web_Demo(  )
```

or any other function that invokes `Ether_Task_Setup_Default`.

If you have a WiFi Wildcard, you can initialize it by typing one of the following at the terminal prompt:

```
WiFi_Task_Setup_Default(  )
```

or:

```
WiFi_Web_Demo(  )
```

or any other function that invokes `WiFi_Task_Setup_Default`. Now to run the serial tunnel demonstration type at your terminal:

```
Tunnel_Test(  )
```

As usual, interactive function calls from the terminal must be typed with no spaces between the function name and the `(` character, and must include at least one space after the `(` character. After typing this command, `Tunnel_Test` is running.

`Tunnel_Test` prints a welcoming statement telling us that it is waiting for us to open a connection using Putty in "Raw" mode. For information about how to download and use the Putty terminal, see the section above titled "Using the Free Putty TCP/IP Terminal for Testing and Development". Briefly, after a simple Google search for "Putty", download the small executable, and double click its icon to start it. Simply type the EtherSmart/WiFi Wildcard's IP address (such as 10.0.1.22) and port 80 into the boxes in the Putty configuration screen, select the "Raw" mode, and click the "Open" button to establish the connection. Type some (optional) text followed by a carriage return in the Putty terminal window to send the first line to the Wildcard.

`Tunnel_Test` waits for an incoming TCP/IP connection on this module's IP address on port 80 (the default local port). The `Ether_Connection_Manager` routine that is running in the Ethernet task automatically stores the linefeed-delimited first line into the `HTTP_Inbuf` counted buffer, and examines the line to see if it starts with "GET ". If the first line starts with GET (which is the request keyword issued by a web browser), the connection manager invokes the web server as explained in the next section. If the first line does not start with GET (as is the case in this example), the `Ether_Passive_Non_Web_Connection` returns a true value and the wait loop ends. The `Tunnel_Test` function in Listing 1-7 then prints the following message to the QEDTerm serial terminal:

```
An incoming connection has been accepted;
each incoming line will be printed to the serial terminal, and a
response prompt will be sent to the ethernet terminal.
To exit this routine, wait 20 seconds without typing a line.
```

The first line has already been loaded into the `HTTP_Inbuf`; an explanation for this behavior is presented in the Web Server section below. `Tunnel_Test` prints the contents of this line to the

QEDTerm serial terminal by fetching the 16-bit count that is stored in the first 2 bytes of **HTTP_Inbuf**, and using **Emit** to print each subsequent character that is stored in the buffer. Note that the characters are stored starting at the **HTTP_Inbuf** address + 2, as the buffer count occupies the first two bytes. Because the buffers are in paged memory, the page-smart fetch routines **FetchInt** and **FetchChar** are used in Listing 1-7 to get the buffer count contents; standard C assignments don't work in paged memory.

The main **do…while** loop in Listing 1-7 calls **Ether_Get_Line** to input a carriage-return delimited line of text into the **Ether_Inbuf**, and invokes a **for** loop to print the buffer contents to the QEDTerm terminal screen. For each incoming line, the "Line was received>" prompt is printed to the Putty terminal window by the **Ether_Send_Buffer** function. **Ether_Send_Buffer** sends a message to the Ethernet task that was set up by **Ether_Web_Demo** or **WiFi_Web_Demo** and the Ethernet task responds by sending a message in the **ether_response** mailbox. The call to **Ether_Await_Response** fetches the contents of and clears the **ether_response** mailbox. The least significant word returned by **Ether_Await_Response** is the number of bytes actually sent by **Ether_Send_Buffer**; in this simple example we discard this value. Note that a more sophisticated program would typically use the more efficient non-blocking function **Ether_Check_Response** to monitor the and clear **ether_response** mailbox.

The **Tunnel_Test** function terminates when **Ether_Get_Line** does not receive any characters within the specified 20 second timeout.

Listing 1-7 Interactive Serial Tunneling Test Function from the Demo Program.

```
// ******************* SERIAL TUNNELING TEST **************

_Q void Tunnel_Test( void )
// works for xport or wiport (ethernet/wifi)
// call after doing Ether_Task_Setup_Default or WiFi_Task_Setup_Default
// Waits for incoming non-web connection on this module's IP address on port 80
// (the default local port), and examines the linefeed-delimited first line to see
// if it starts with GET; if not, it accepts it as a passive non-web connection.
// So: don't type GET as the first characters (you'll confuse the system).
// The easiest way to open a connection is via Putty, using the "raw" mode.
// Note: If you use "telnet" mode, you'll see garbage characters in the first line;
// these are control bytes with msbit set that are accepted by the connection manager.
{  int num_received=0, inbufcount=0, i=0, numlines=0;
   char c;
   int timeout = 20000; // 20 second timeout for testing; sets delay til function exit
   xaddr ether_inbuf_base = Ether_Inbuf(E_MODULENUM);
   uint ether_inbuf_size =  Ether_Inbufsize(E_MODULENUM);
   xaddr http_inbuf_base = HTTP_Inbuf(E_MODULENUM);
   char* prompt = "Line was received>";
   int prompt_length = strlen(prompt);
   printf("\rWaiting for connection (type a carriage return from QEDTerm\
        to abort)...\r");
   printf("Suggestion: To connect, use Putty in 'raw' mode, specify IP address,\
     port 80,\r");
   printf("and type a carriage return once the session window opens.\r");
   do
   {  PauseOnKey;     // abort on CR, pause/resume on other keys
      Ether_Check_Response(E_MODULENUM);  // keep mailbox clean, ignore messages
   }
   while( !Ether_Passive_Non_Web_Connection(E_MODULENUM) ); // wait for connection
   printf("\rAn incoming connection has been accepted;\r");
   printf("each incoming line will be printed to the serial terminal, and a\r");
```

```
    printf("response prompt will be sent to the ethernet terminal.\r");
    printf("To exit this routine, wait 20 seconds without typing a line.\r");
       // the first line is in HTTP_Inbuf (because webserver had to check identity)
       // count is in 1st 2bytes of buffer
    inbufcount = (FetchInt(http_inbuf_base)); // count stored in 1st 2bytes of buffer
    for( i=0; i<inbufcount; i++)
       {  c = FetchChar(http_inbuf_base + 2 + i); // skip 2byte count,get char
          Emit(c);
       }  // type first line including terminating crlf to local serial terminal
    do    // loop and echo additional lines til timeout...
    {  Ether_Get_Line( ether_inbuf_base,ether_inbuf_size,CR_ASCII,1,1,timeout,
                        E_MODULENUM);
             // xlbuf,maxchars,eol,discard.alt.eol?,no.msbitset?,timeout_msec,module
             // get 1 line into counted ether_inbuf (count is stored in first 2 bytes)
       num_received = (int) Ether_Await_Response(E_MODULENUM);  // get lsword of result

       inbufcount = (FetchInt(ether_inbuf_base));
       for( i=0; i<inbufcount; i++)
          {  c = FetchChar(ether_inbuf_base + 2 + i); // skip 2byte count,get char
             Emit(c);
          }  // type line including terminating crlf to local serial terminal
       if(num_received)  // send prompt and wait for response if chars were received...
       {  Ether_Send_Buffer( STRING_XADDR(prompt),prompt_length,timeout,E_MODULENUM);
          Ether_Await_Response(E_MODULENUM);  // wait for result, ignore rtn value
          numlines++;                    // bump numlines
       }
    }
    while(num_received);    // until no chars rcvd due to timeout or connection close
    Ether_Disconnect_Flush(E_MODULENUM);   // clean up
    Ether_Await_Response(E_MODULENUM);     // synchronize to result mailbox before exit
    printf("\r\nConnection terminated.\r\n");
 }
```

Serial Tunneling Buffer Management Functions

Communicating via Ethernet requires the use of buffers to hold incoming and outgoing data. The relevant functions are listed in Table 1-6.

Table 1-6 Buffer Management Functions for Serial Tunneling.

Cat	Ether_Outbuf
ETHER_BUFSIZE_DEFAULT	Ether_Outbuf_Cat
Ether_Inbuf	Ether_Outbufsize
Ether_Inbufsize	Ether_Set_Inbuf
ETHER_MIN_BUFFER_SIZE	Ether_Set_Outbuf

As described in the "EtherSmart/WiFi Driver Data Structures: The LBuffer, a Counted Buffer" section above, functions that manipulate buffers need to know the number of valid bytes stored in the buffer. A convenient place to store this information is in the buffer itself. For this reason, this driver code uses a data structure called an "LBuffer", defined as a buffer with a 16-bit count stored in the first two bytes of the buffer, with the data following.

In general, the buffers are in paged RAM. C assignment operators do not work in paged memory, so the operating system's store and fetch routines as declared in the `memory.h` file in the Mosaic include directory must be used to access data stored in the buffers. These functions include `FetchChar`, `FetchInt`, `StoreChar`, and `StoreInt`.

The input and output buffers for serial tunneling (and email) via the EtherSmart/WiFi Wildcard are called **Ether_Inbuf** and **Ether_Outbuf**, respectively. These functions each return the base 32-bit extended address (xaddress) of the LBuffer. Each of these buffers has a maximum size that is set at initialization time. The default size of **Ether_Inbuf** and **Ether_Outbuf** is given by the constant **ETHER_BUFSIZE_DEFAULT**. The functions that write to the buffers such as **Cat** and **Ether_Outbuf_Cat** accept the maximum buffer size as an input parameter, and ensure that they never write beyond the allowed buffer size. If you need specify a new buffer base xaddress or size that is different from the values set at initialization time, use the functions **Ether_Set_Inbuf** and **Ether_Set_Outbuf**. For complete descriptions of each of these functions, consult the EtherSmart/WiFi Wildcard Glossary document.

For example, a function like **Ether_Send_LBuffer** expects the 32-bit extended address (xaddress) of the LBuffer as an input parameter, and sends the contents on the active TCP/IP connection. This function automatically fetches the number of data bytes in the buffer (its "count") from the buffer's first 2 bytes, and sends data starting at the specified xaddress+2. Receiving functions such as **Ether_Get_Data**, **Ether_Get_Chars** and **Ether_Get_Line** expect the 32-bit extended address (xaddress) of an LBuffer as an input parameter, and receive incoming data to the buffer, storing the count at the specified xaddress, and storing the data starting at xaddress+2. The glossary entry for each buffer handling function tells whether it is dealing with an uncounted buffer or an LBuffer.

Rule for Accessing Data in EtherSmart/WiFi Buffers and Data Structures

Because the EtherSmart/WiFi buffers and data structures are in paged memory, the page-smart routines **FetchInt, FetchChar, StoreInt** and **StoreChar** are used to access them; standard C assignments don't work in paged memory.

Serial Tunneling Data Transmission and Reception Functions

The essence of serial tunneling is an exchange of binary or ASCII bytes between two computers via a TCP/IP connection. Table 1-7 lists the transmission and reception functions; these work for both the EtherSmart and WiFi Wildcards.

Table 1-7 Transmission and Reception Functions for Serial Tunneling.

Ether_Add_Chars	**Ether_Get_Line**
Ether_Add_Data	**Ether_Send_2Buffers**
Ether_Add_Line	**Ether_Send_Buffer**
Ether_Get_Chars	**Ether_Send_LBuffer**
Ether_Get_Data	

The data transmission function **Ether_Send_LBuffer** sends a counted LBuffer out on the active TCP/IP connection. It accepts the 32-bit base xaddress of an LBuffer, a timeout count in milliseconds (msec), and a modulenum as input parameters. Recall that an LBuffer stores the number of valid bytes in the first 2 bytes of the LBuffer, with the data following. **Ether_Send_Buffer** accepts the 32-bit base xaddress of an uncounted buffer, the buffer count (number of bytes to be sent), a timeout in msec, and the modulenum. It sends the specified buffer contents out on the active TCP/IP connection. **Ether_Send_2Buffers** sends the contents of two uncounted buffers sequentially, each with a specified base xaddress and count. See the glossary entries of these functions for additional details.

Each of these transmission functions requires that the Ethernet task running the **Ether_Service_Loop** is active, as set up by **Ether_Task_Setup** or **Ether_Task_Setup_Default** for the EtherSmart Wildcard, or by **WiFi_Task_Setup** or **WiFi_Task_Setup_Default** for the WiFi Wildcard. The transmission functions each send a message via the **ether_command** mailbox to the Ethernet task, which then performs the requested action and writes a response to the **ether_response** mailbox. The contents of this mailbox must be checked and cleared by the application program using either the non-blocking **Ether_Check_Response** function, or the blocking **Ether_Await_Response** function. The least significant 16-bit word of the result that is returned by any of these functions is the number of bytes actually sent by the transmission function.

The actions performed by the data reception functions are easy to understand once the naming terminology is defined. These functions either “Add” bytes, appending them to a buffer that may already contain data, or “Get” bytes, storing them starting at the beginning of the buffer. The reception functions can receive “Data” which is a binary stream of bytes, or line-oriented ASCII “Chars”. A “Line” is a set of ASCII “Chars” ending with a specified “end of line” (eol) delimiter such as a linefeed (ASCII 0x0A) or carriage return (ASCII 0x0D).

Note that it is permissible to use the **Ether_Add_Data** or **Ether_Get_Data** functions for either binary or ASCII data, as long as no line-oriented decision making is required during the reception process. However, the ASCII character oriented routines should not be used for binary data, as binary characters that happen to equal the specified “eol” character would get special treatment that could lead to unexpected results. The recommended course of action is to use functions that contain the sub-words “Chars” or “Line” for ASCII streams, and use the functions that contain the sub-word “Data” for binary streams.

The **Ether_Add_Data** function receives a binary stream of bytes into a counted LBuffer; as usual, it requires that the Ethernet task is running. Recall that an LBuffer stores the count in its first 2 bytes, with the data following. **Ether_Add_Data** accepts as input parameters the 32-bit base xaddress of an LBuffer, the maximum number of bytes to be added to the buffer, a timeout parameter in units of milliseconds, and the modulenum of the specified Wildcard. It fetches the starting count from the LBuffer, adds it to the specified buffer base xaddress+2, and uses the resulting xaddress as the location where the first byte of incoming data will be stored. The data input operation stops if the amount of data in the specified buffer (including any prior data, but not including the 2-byte count) exceeds the specified maximum-number-of-bytes parameter. The function terminates when either the specified number of bytes has been received, or the specified timeout is reached, whichever comes first. Before exiting, the function writes the updated count into the first 2 bytes of the LBuffer.

The non-appending `Ether_Get_Data` function accepts the same input parameters as `Ether_Add_Data`. `Ether_Get_Data` simply writes a 16-bit zero to the count in the first 2 bytes of the specified LBuffer, and calls `Ether_Add_Data`. The result is that the first byte of data is stored at the specified LBuffer xaddress+2.

The most configurable reception function for line-oriented ASCII streams is `Ether_Add_Chars`. Its function prototype and glossary entry are as follows:

```
void Ether_Add_Chars ( xaddr xlbuffer, uint maxbytes, uint maxlines, char eol,
        int discard_alt_eol, int discard_msbit_set, uint timeout_msec, int modulenum)
```

This function sends a message to the Ethernet task which requests and stores incoming ASCII data from the specified EtherSmart/WiFi modulenum. The data is appended to `xlbuffer`, a counted buffer whose byte count is stored in the first 2 bytes of the buffer, and the count is incremented by the number of appended bytes. The `xlbuffer` parameter is a 32-bit extended address that holds the 16-bit buffer count followed by the buffer data. The appended data is stored starting at `xlbuffer+2+prior_count`, where `prior_count` is the 16-bit contents at `xlbuffer` upon entry into this routine. The data input operation stops if the amount of data in the specified buffer (including any prior data, but not including the 2-byte count) exceeds the specified `maxbytes` parameter. A maximum of `maxlines` are accepted, where a “line” is a data sequence ending in the specified `eol` (end of line) character. If the `maxlines` input parameter = -1, then the line limit is ignored. If `maxlines` = 0, then all except the last incoming line are discarded, and only the last line is added to the buffer, excluding the final `eol` character which is discarded and not added to the buffer. The `eol` parameter is a single character that specifies end of line. Typical values are ‘CR’ = 0x0D or ‘LF’ = 0x0A. Dual-character eol sequences such as CRLF are not allowed. If `eol` = ‘LF’, we define the “alternate eol” is a ‘CR’. For all other eol chars (including a ‘CR’), the “alternate eol” is a ‘LF’. If the `discard_alt_eol` flag parameter is true, the alternate to the specified `eol` character is discarded/ignored by this routine. If the flag is false, the alternate eol char does not get special treatment, and is stored in the buffer like any other character. If the `no_msbitset` flag parameter is true, then any characters having its most significant (ms) bit set (bit7, bitmask = 0x80) is discarded and is not stored in the buffer. This is useful, for example, if the incoming data is sent by a Telnet application; some configuration data is transmitted that can be filtered out by discarding characters with their ms-bits set. This function exits within the specified `timeout_msec` whether or not the maximum number of bytes or lines have been accepted. When the action function dispatched by the Ethernet task has completed, a response comprising the command byte in the most significant byte, module number in the next byte, and `numbytes_appended` in the remaining 2 bytes is placed in the `ether_response` mailbox. To test for a buffer overrun, fetch the 2-byte count from `xlbuffer` and test whether it is greater than or equal to the allowed `maxbytes`. After calling this routine the application must clear the `ether_response` mailbox using `Ether_Check_Response` or `Ether_Await_Response`.

The non-appending `Ether_Get_Line` function simply zeros the count at the specified LBuffer and calls `Ether_Add_Line`. As a result, `Ether_Get_Line` stores characters starting at the beginning of the buffer instead of appending it after any prior buffer characters.

While this may sound complicated, using these functions is actually quite straightforward. A simple example extracted from Listing 1-7 illustrates how to use `Ether_Get_Line`:

```
int timeout = 20000; // 20 second timeout for testing; sets delay til function exit
```

```
xaddr ether_inbuf_base = Ether_Inbuf(E_MODULENUM);
uint ether_inbuf_size =  Ether_Inbufsize(E_MODULENUM);
Ether_Get_Line( ether_inbuf_base,ether_inbuf_size,CR_ASCII,1,1,timeout,
                    E_MODULENUM);
            // xlbuf,maxchars,eol,discard.alt.eol?,no.msbitset?,timeout_msec,module
            // get 1 line into counted ether_inbuf (count is stored in first 2 bytes)
num_received = (int) Ether_Await_Response(E_MODULENUM);  // get lsword of result
```

In this code fragment, we first initialize some variables that are passed as parameters to **Ether_Get_Line**. One of the rules of calling operating system or kernel extension functions is that no function nesting is allowed: it is illegal to invoke one of these functions within the parameter list of another. By capturing the results returned by the **Ether_Inbuf** and **Ether_Inbufsize** functions into variables, we avoid the nesting prohibition when calling **Ether_Get_Line**. The **Ether_Get_Line** function is invoked to place the received characters into the counted LBuffer **Ether_Inbuf**, with a maximum number of received characters limited to **Ether_Inbufsize** (default value = 510 set at initialization time). The end of line character is specified as the ASCII carriage return (0x0D). Note that the standard line ending sequence for most standard TCP/IP text transfers is carriage return followed by a linefeed (0x0D0A). We pass a true (nonzero) flag as the **discard_alt_eol** parameter to specify that the alternate end of line character (the linefeed) should be not placed into the buffer. We pass a true flag as the **no_msbitset** parameter to specify that any characters with their most significant bit set should not be stored in the buffer. The 20,000 msec timeout and modulenum complete the input parameter list. **Ether_Get_Line** stores the input parameters in the **Ether_Info** structure and sends a message in the **ether_command** mailbox to the Ethernet task which performs the requested action and places the response in the **ether_response** mailbox. We then invoke **Ether_Await_Response** to poll and clear the **ether_response** mailbox. The number of bytes received is returned in the least significant 16 bits of the result, and we load this into the **num_received** variable for use in the program as shown in Listing 1-7.

Rule for Calling Operating System Functions

When calling operating system or kernel extension functions, no function nesting is allowed: it is illegal to invoke one of these functions within the parameter list of another.

Serial Tunneling Connection Functions

The pre-coded EtherSmart/WiFi Wildcard software provides routines to connect to a remote computer, monitor the status of a connection, disconnect, and optionally flush bytes that are in the input buffers. Table 1-8 summarizes these functions.

Table 1-8 Connection Functions for Serial Tunneling.

Ether_Connect	**Ether_Disconnect_Flush**
Ether_Connect_Status	**Ether_Flush**
Ether_Disconnect	**Ether_Flush_NBytes**
Ether_Disconnect_During_Send	**Ether_Passive_Non_Web_Connection**

A serial tunneling connection may be established either at the request of a remote computer (an "incoming connection request"), or by the EtherSmart/WiFi Wildcard itself (an "outgoing connection request"). Only one connection may be active at a time; this limitation is set by the Lantronix device. If there is no active connection, any incoming connection request directed at the proper local IP address and local port (port 80 by default) will be accepted by the Lantronix hardware. There is no way to change the local port number for incoming connections "on the fly"; see the section titled "Configuring the Lantronix Device" for more details.

When an incoming connection is accepted by the EtherSmart/WiFi Wildcard, the **Ether_Connecion_Manager** called by the **Ether_Service_Loop** in the Ethernet task updates the connection status. Your application program can monitor the status of the connection by calling **Ether_Connect_Status**. Table 1-9 summarizes the meaning of the status parameter returned by this function. A zero value means that there is no active connection. Odd values indicate transient conditions as summarized in the table. A value of 2 means that we have successfully initiated an outgoing connection to a remote computer. A value of 4 means that we have accepted an incoming connection, and the connection manager has identified it as non-web, while a value of 6 means that an incoming connection has been accepted and has been identified as coming from a web browser. To identify the type of the incoming connection, the connection manager looks to see if the first characters of the first line are "GET " (without the quotes); a web browser starts its request with the uppercase GET substring. If GET is found, the connection is identified as HTTP/web; otherwise, it is identified as non-web.

Table 1-9 Values Returned by the Ether_Connect_Status function.

Value	Meaning
0	Not connected.
1	Pending disconnect (disconnect failed, waiting to retry the disconnect)
2	Active connection initiated by us
3	Active connection that has been interrupted by a remote disconnect during a send
4	Passive connection initiated by remote, identified as non-HTTP
5	Passive non-HTTP connection interrupted by a remote disconnect during a send
6	Passive connection initiated by remote, identified as HTTP (GET was detected)
7	Passive HTTP connection interrupted by a remote disconnect during a send
8	Passive HTTP connection with web service completed, awaiting connection close

The good news is that, in nearly all cases, you don't have to worry about the exact numerical values in Table 1-9. A simple zero-versus-nonzero check of the **Ether_Connect_Status** return value tells the application program whether a connection is present or not. The application program does not have to test for a web connection because web connections are typically handled "in the background" without the need for intervention by the application program. The useful function **Ether_Passive_Non_Web_Connection** returns a true (nonzero) flag when the **Ether_Connect_Status** is 4. Thus, calling **Ether_Passive_Non_Web_Connection** is a good way to monitor for an incoming serial tunneling request from a remote computer.

The **Ether_Disconnect_During_Send** function returns a true value when the remote computer disconnected while we were sending data. This low-level function is typically not used in application programs.

To establish an outgoing connection from the EtherSmart/WiFi Wildcard, call **Ether_Connect**, passing it the destination IP address and remote port, a timeout in milliseconds, and the modulenum. Its function prototype is:

```
void Ether_Connect( char ip1, char ip2, char ip3, char ip4, int port,
               int timeout_msec, int modulenum )
```

For example, to connect EtherSmart/WiFi Wildcard modulenum 3 to a remote host with IP address 10.0.1.145 on port 23, allowing up to 15 seconds for the connection to take place, execute:

```
Ether_Connect( 10, 0, 1, 145, 23, 15000, 3);
```

If there is no current connection (remember that the Lantronix device can only implement one connection at a time), this function will try to connect to the specified remote for up to the specified 15 seconds. If there is already an existing connection when this function is called, no connection attempt will be made. You can invoke **Ether_Connect_Status** to check whether there is an active connection. As with nearly all of the serial tunneling functions, the **Ether_Connect** function dispatches a message in the **ether_command** mailbox to the Ethernet task which performs the requested action and returns a response in the **ether_response** mailbox. The application program must monitor and clear the response mailbox using either **Ether_Check_Response** or **Ether_Await_Response**. The least significant 16 bits returned by one of these functions after calling **Ether_Connect** contains an error flag. The error flag is zero if the connection attempt succeeded, and nonzero if it failed. See the glossary entry for **Ether_Error** for a description of the error code values.

To perform a simple disconnect, pass the specified EtherSmart/WiFi modulenum parameter to **Ether_Disconnect**. To disconnect and "flush" (discard) any incoming bytes from the input buffers, execute **Ether_Disconnect_Flush**. Just as with **Ether_Connect**, the least significant 16 bits returned by the next invocation of **Ether_Check_Response** or **Ether_Await_Response** yields the error result for the operation.

You can also flush the input buffers at any time using the **Ether_Flush** and **Ether_Flush_NBytes** functions. **Ether_Flush** is the most thorough flush; it waits up to 0.25 seconds since the last discarded byte and, if no additional byte becomes available in that time, exits. The maximum execution time of this routine is 8 seconds, even if data is being continually flushed during this time; this prevents an indefinite "hung" state. **Ether_Flush_NBytes** accepts a number of bytes and a timeout parameter, and flushes up to the specified number of bytes from the input buffer on the EtherSmart/WiFi Wildcard.

Serial Tunneling Inter-Task Service Management Functions

The serial tunneling services are provided by means of the Ethernet task that is set up at initialization time by **Ether_Task_Setup** or **Ether_Task_Setup_Default** on the EtherSmart Wildcard, or by **WiFi_Task_Setup** or **WiFi_Task_Setup_Default** on the WiFi Wildcard. Your application program invokes the driver functions, most of which work by sending a message in the

ether_command mailbox to the Ethernet task which performs the requested action. The presence of a dedicated Ethernet task provides a means to rapidly service communications events such as incoming connection requests, requests from a web browser, and communications to and from the application task. Table 1-10 lists the Ethernet driver functions that manage the inter-task communications.

Table 1-10 Inter-Task Service Management Functions.

Ether_Await_Response	**Ether_Ready_For_Command**
Ether_Check_Response	**Ether_Service_Loop**
Ether_Command_Manager	**ether_service_module**
Ether_Connection_Manager	**Ether_Tunnel_Enable_Ptr**
Ether_Error	**HTTP_Enable_Ptr**
Ether_Error_Clear	**HTTP_Server**

As explained in the "EtherSmart/WiFi Driver Data Structures" section above, a "task" is an environment that is capable of running a program. The task contains a user area, stacks, and buffers that enable a program to run independently of other tasks. The multitasking operating system switches between tasks, providing timely provision of a variety of services. A "mailbox" is a 32-bit variable in common memory that conveys a message from one task to another. A mailbox must be cleared before a new message may be sent; this helps to synchronize the tasks and ensures that no messages are discarded without being read by the receiving task. A mailbox may be read and cleared by a "blocking" function that waits until the mailbox contains a message (that is, until it contains a non-zero value) and then reads out, reports, and zeros the mailbox. It may also be checked by a "non-blocking" function that reads and reports the contents of the mailbox and, if it contains a non-zero value, clears it. Non-blocking mailbox reads are often preferred because they avoid tying up one task while waiting for another task to send a message.

The EtherSmart/WiFi driver code uses mailboxes to coordinate the provision of communications services by the Ethernet task to the programmer's main application task. This is all done transparently, so you don't have to be an expert on inter-task communications to use the EtherSmart/WiFi Wildcard. Three 32-bit mailboxes named **ether_command**, **ether_response**, and **ether_gui_message** are allocated in common RAM when the initialization routine executes. For example, when the application program invokes a command such as **Ether_Get_Data**, a message is dispatched via the **ether_command** mailbox to the Ethernet task. Consequently, the Ethernet task gets the incoming data and places it in the specified buffer. The Ethernet task then sends the number of bytes received as the least significant 16 bits in the **ether_response** mailbox. The most significant 16 bit word of the mailbox contains the command byte and the modulenum. This mailbox must be cleared by the application program using either the blocking function **Ether_Await_Response**, or the non-blocking function **Ether_Check_Response**. The main loop of your program's application task should periodically call one of these two functions to manage the interactions with the Ethernet task.

If an error occurs during the execution of a function by the Ethernet task, a 16-bit error variable in the **Ether_Info** structure is set. The application task can read this variable using the

Ether_Error function, and can clear it using the **Ether_Error_Clear** function. Some functions also return the error value as the least significant 16 bits in the **ether_response** mailbox.

The **Ether_Service_Loop** is a simple infinite loop. In the loop body are calls to **Ether_Connection_Manager**, **Ether_Command_Manager**, and **Pause**. The default version of **Ether_Service_Loop** specifies the contents of the **ether_service_module** variable as the parameter passed to **Ether_Connection_Manager** and **Ether_Command_Manager** in the infinite loop body. The **ether_service_module** variable is set by the initialization functions **Ether_Init** and its calling functions (**Ether_Setup**, **Ether_Setup_Default**, and **Ether_Task_Setup**). It specifies which Wildcard module is accessed by the **Ether_Service_Loop** routine running in the Ethernet control task. Because each of these variables is automatically initialized by higher level functions, you typically don't have to worry about them. You can even have two EtherSmart and/or WiFi Wildcards installed, one for revectored serial, and the other providing communications services (serial tunneling, email, and web service).

When an incoming connection is accepted by the Wildcard, the **Ether_Connecion_Manager** called by the **Ether_Service_Loop** in the Ethernet task updates the connection status. Your application program can monitor the status of the connection by calling **Ether_Connect_Status** as described in the prior section. Incoming connection requests are accepted by the Lantronix device if the connection is directed to the correct local IP address and local port. The default local port is (decimal) 80, which is the standard port used by web browsers to request a web connection. We use port 80 as the local port so that web connections can be accepted, but in many applications we also want to be able to detect and service incoming serial tunneling connections.

If the variable pointed to by **HTTP_Enable_Ptr** is true, the **Ether_Connection_Manager** automatically stores the linefeed-delimited first line into the **HTTP_Inbuf** counted buffer, and examines the line to see if it starts with "GET ". If the first line starts with GET (which is the request keyword issued by a web browser), the connection manager invokes the **HTTP_Server** to serve out the requested web page as explained in the next section. If the first line does not start with GET and if the variable pointed to by **Ether_Tunnel_Enable_Ptr** is true, then **Ether_Passive_Non_Web_Connection** returns a true value.

The application task can use the control variables accessed pointed to by **Ether_Tunnel_Enable_Ptr** and **HTTP_Enable_Ptr** to direct this process of accepting incoming connections. After execution of any of the initialization functions, both of these variables are true by default.

If your application will never use the dynamic webserver function, you can simplify the handling of incoming serial tunneling connections by setting the **HTTP_Enable_Ptr** variable to zero. This disables the automatic loading of the first linefeed-delimited line of an incoming connection into **HTTP_Inbuf**, allowing the application program to manage the entire reception process.

If your application will use the dynamic webserver but will never need to accept incoming serial tunneling connections, you can zero the variable pointed to by **Ether_Tunnel_Enable_Ptr**. In most cases this will not affect performance, but in some extreme cases it may increase the robustness of error recovery in handling web requests.

Sending Email

The EtherSmart/WiFi Wildcard enables your application program to send emails to other computers on the network when significant events occur. The driver software implements SMTP (Simple Mail Transfer Protocol) to send outgoing email to a mail server on the LAN (Local Area Network). Outgoing emails can be used by your instrument to send alerts or status updates.

The **`Ether_Send_Email`** function handles all of the details for you. You pass to the function the gateway IP address on your LAN, and strings that specify the sender email address, recipient email address, and the email body including an optional subject line. The EtherSmart/WiFi driver software and the mail server running on the gateway computer cooperate to process the email.

Note that the Lantronix device can connect only to destinations on the local network. If you need to send an email to a destination that is not on the LAN or WLAN (Wireless LAN), address the email to the gateway (router) computer's IP address, and ask your system administrator to configure the gateway to "relay" (forward) the email to the desired destination. Also, note that the EtherSmart/WiFi Wildcard can send emails, but cannot receive them. To send a message to the Wildcard, use a serial tunneling connection or a forms-based web connection as described in other sections of this document.

A Demonstration Email Function

Listing 1-8 illustrates the **`Email_Test`** function from the demo program. Let's take a look at how this function works. If you have not already done so, initialize the Wildcard by invoking an initialization function. If you have an EtherSmart Wildcard, you can type one of the following at the terminal prompt:

```
Ether_Task_Setup_Default(  )
```

or:

```
Ether_Web_Demo(  )
```

or any other function that invokes **`Ether_Task_Setup_Default`**.

If you have a WiFi Wildcard, you can initialize it by typing one of the following at the terminal prompt:

```
WiFi_Task_Setup_Default(  )
```

or:

```
WiFi_Web_Demo(  )
```

or any other function that invokes **`WiFi_Task_Setup_Default`**. Now to run the email demonstration type at your terminal:

```
Email_Test(  )
```

As usual, there must be no space before the **`(`** and at least one space before the **`)`** character. An examination of Listing 1-8 reveals how easy it is to send an email. Note that the hostname, sender, recipient, and IP address must be edited and the demo program compiled and downloaded for this function to work on your local network.

The four string definitions at the top of Listing 1-8 specify the hostname, email sender, email recipient, and the email body, respectively. Edit these to represent values that make sense on your LAN. Note that the first line of the **`email_body_str`** is:

```
Subject: EtherSmart Email Test\r\n\
```

This is the optional subject line. As shown, the word **`Subject:`** should be the first word on the line. The **`Ether_Send_Email`** routine automatically prepends a **`To:`** line to the email body, emplacing the recipient name string that is passed as an input parameter. You can include other “email header” items in the body of your email, such as:

```
Date:
Cc:
From:
```

Each of these lines should be terminated with a carriage return/linefeed (**`\r\n`**) which is the standard end of line sequence for the SMTP protocol.

Listing 1-8 Interactive Email Test Function.

```
// ********************* EMAIL  TEST ***************************

// NOTE: Edit these to specify valid host, sender, and recipient:
char* hostname_str = "wildcard.yourdomain.com";   // hostname string
char* sender_str =  "niceguy@yourdomain.com";  // email sender string
char* recipient_str = "notso_niceguy@yourdomain.com";   // email recipient string

char* email_body_str =  // email_body string
"Subject: EtherSmart/WiFi Email Test\r\n\
This is a test email from the EtherSmart/WiFi Wildcard.\r\n\
Emails can of course have multiple lines...";

#define SMTP_PORT 25  // email Simple Mail Transfer Protocol destination port

_Q int Email_Test( void )
// works for xport or wiport (ethernet/wifi)
// call after doing Ether_Task_Setup_Default or WiFi_Task_Setup_Default
// smtp email test routine.
// Note: YOU MUST EDIT THIS FUNCTION's HARD-CODED IP ADDRESS
// AND the hostname_str, sender_str, and recipient_str BEFORE CALLING!
// Otherwise, it will not work on your system.
// error= 0 on success, or error_no_response = 0x10, or smtp 3-digit error code
{  xaddr scratchbuf = Ether_Inbuf( E_MODULENUM );     // scratchpad xbuffer
   Ether_Send_Email(
      STRING_XADDR(email_body_str), strlen(email_body_str), // subject plus email
      STRING_XADDR(hostname_str), strlen(hostname_str),
      STRING_XADDR(sender_str), strlen(sender_str),
      STRING_XADDR(recipient_str), strlen(recipient_str),
      scratchbuf,       // scratchpad xbuffer
      10, 0, 1, 1,      // IMPORTANT: ENTER YOUR GATEWAY'S IP ADDRESS HERE!
      SMTP_PORT,        // port 25 = standard mailserver destination port
      10000,            // use a 10 second timeout (10,000 msec) for testing
      E_MODULENUM);     // specify module number (must match hardware jumpers)
   return((int) Ether_Await_Response(E_MODULENUM)); // error code is in lsword
}
```

The **`Email_Test`** function in Listing 1-8 simply sets up the input parameter list and calls **`Ether_Send_Email`**. See the “EtherSmart/WiFi Driver Data Structures: Passing String Extended Addresses as Function Parameters” section above for a discussion of the **`STRING_XADDR`** macro that converts a 16-bit string address to a full 32-bit xaddress as required by **`Ether_Send_Email`**. The four lines after the function invocation specify the string xaddress and count for the subject, host name, sender, and recipient, respectively.

In this example the **`scratchbuf`** local variable is defined as an xaddress that is initialized to the recommended email scratchpad buffer **`Ether_Inbuf(E_MODULENUM)`**. As discussed in earlier sections, it is illegal to nest operating system function calls inside a parameter list of another operating system function call. The definition and passing of **`scratchbuf`** enables us to pass the **`Ether_Inbuf`** xaddress without violating this rule. The specified ram scratchpad buffer holds SMTP strings going to and from the Wildcard. The recommended minimum scratchpad buffer length is 128 bytes; it must be able to hold the sender, recipient, and hostname strings plus sixteen extra bytes. The email body is not transferred to the scratchpad buffer, so you can define an email body that is significantly longer than the scratchpad buffer.

This example shows a static email body, but you can create the email body "on the fly" in your application program. To dynamically build up the email string in the **`Ether_Outbuf`** buffer, use the **`Ether_Outbuf_CAT`** function to build the email body one line at a time. If you put the email body in **`Ether_Outbuf`**, pass the buffer xaddress+2 as the string xaddress, and the contents of the buffer xaddress as the count. This is because **`Ether_Outbuf`** is an LBuffer with the 16-bit count stored in the first two bytes, followed by the data in the remainder of the buffer. Because the buffer is typically in paged (as opposed to common) memory, you must use the operating system function **`FetchInt`** to extract the count from the buffer. In the example above, using **`Ether_Outbuf`** instead of **`email_body_str`** would involve declaring the new variables:

```
uint ether_outbuf_lcount;
xaddr ether_outbuf_xaddr = Ether_Outbuf(E_MODULENUM);
```

After the email body is loaded into the **`Ether_Outbuf`** using **`Ether_Outbuf_Cat`**, the following assignment can be performed:

```
ether_outbuf_lcount = FetchInt( ether_outbuf_xaddr);
```

Finally, in Listing 1-8, to use **`Ether_Outbuf`** for the email body you would replace the line:

```
STRING_XADDR(email_body_str), strlen(email_body_str),
```

with the line:

```
ether_outbuf_xaddr+2, ether_outbuf_lcount,
```

to specify the email body xaddress and count.

After the strings and scratchpad buffer in the parameter list in Listing 1-8, the next four parameters specify the destination IP address to which the email is directed. Often this is the IP of the gateway computer on the LAN, as the gateway often runs the mail server. Make sure to edit the demo code to specify a valid IP address on the LAN.

The next parameter is the destination TCP/IP port number. In most cases this will be port 25, the SMTP port as defined by the constant in the demo program.

The next parameter in the **`Ether_Send_Email`** invocation is a timeout in units of milliseconds. It is important not to make this too small, as there can be significant delays on a network. The example shows a 10 second (10,000 millisecond) timeout, but the value required on your network may be different.

The final parameter passed to **Ether_Send_Email** is the modulenum.

When the application program invokes **Ether_Send_Email**, a message is dispatched via the **ether_command** mailbox to the Ethernet task. Consequently, the Ethernet task opens a connection to the specified remote computer, sends the email, and closes the connection. If there is a failure code returned while the email "chat" is occurring, this 3-digit decimal error code is returned and the email transaction is halted. The connection is closed before this routine exits; only 1 email is deliverable per connect. The Ethernet task then sends the result error code in the **ether_response** mailbox. This mailbox must be cleared by the application program using either the blocking function **Ether_Await_Response**, or the non-blocking function **Ether_Check_Response**. The main loop of your program's application task should periodically call one of these two functions to manage the interactions with the Ethernet task. In the example in Listing 1-8, the return statement calls **Ether_Await_Response** and returns the 16-bit error code.

Using the Image Converter to Manage Long Strings

If you need to create long static email body strings, use the "Image Converter" program that is part of your Mosaic development environment. This program converts one or more files, each containing a single string (or other resource) into a named 32-bit base xaddress and count that can be used by your application program. Simply place each string into a separate file with the file extension **.str** or **.html.** The file name and extension (with the dot replaced by an underscore) will become the symbol name. All files to be converted should be in a single directory. Make sure you use C-compatible filenames that include only alphanumeric and _ (underscore) characters, and do not start with a numeral. Invoke the Image Converter from your development environment. In the Image Converter control panel, select the controller platform, and check the "Web files" box. If you are programming in Forth, click the "Advanced" menu and select Forth as the programming language. In the "Directory" area of the control panel, select the directory that contains the specified file(s). Then click on "Convert Files Now".

A pair of files named **image_data.txt** and **image.headers.h** will be created by the Image Converter. The **image_data.txt** contains S-records and operating system commands that load the string image into flash memory on the controller. Because flash memory is nonvolatile, you only need to download this file once to the controller; it need not be reloaded until you change the resources. The **image_headers.h** file declares the 32-bit xaddress and count (size) constants to be used by your application program.

For example, let's say that you have created a rather lengthy email body and stored it in a file named **mail_body.str** where the **.str** file extension indicates that the file contains a string. The **image_headers.h** file declares the following two macros with the appropriate numeric xaddress and count values:

```
#define MAIL_BODY_STR_XADDR 0x700360
#define MAIL_BODY_STR_SIZE 0x98E
```

You can then use these macros as parameters passed to the `Ether_Send_Email` function to specify the email body string. The use of the Image Converter is further described in the next section and in Appendix A.

Loading Your Program

See Appendix A for a discussion of how to load the resources and the kernel extension library files onto the controller board before loading your custom application program.

Introduction to the Dynamic Webserver

The EtherSmart/WiFi Wildcard implements a dynamic webserver that accepts connections from your web browser, serving out static or dynamic web pages. You can “browse into” your instrument using a web browser running on an online PC to monitor the status of your instrument.

Most web pages are “static”, meaning that their content does not change with each visit to the page. However, in the context of embedded computers, “dynamic” web pages that provide up-to-date status information about the computer’s state (inputs, outputs, memory contents, etc.) are very useful. The pre-coded driver functions enable you to serve both static and dynamic web pages.

By coding web content into your application, you can enable remote monitoring and control of your instrument from your web-connected PC. The “embedded web server” that runs when you execute the EtherSmart/WiFi driver code responds to information requests from your browser. You can create a set of web pages, each with a specified name, or “URL” (Uniform Resource Locator) and an associated “handler function” that serves out the static or dynamic web content for the requested page.

The EtherSmart/WiFi Wildcard’s dynamic webserver should not be confused with the Lantronix built-in static webserver that is used for configuration. As described in the “EtherSmart/WiFi Initialization, Configuration and Diagnostics” section, an additional “configuration webserver” is built into the Lantronix device for low-level configuration of the Lantronix firmware, and is available at TCP/IP port 8000. This configuration webserver is typically not used in conjunction with the EtherSmart/WiFi Wildcard, and should not be confused with the dynamic webserver discussed in this section.

Web Service Basics

Most web content is authored using “HTML” which stands for HyperText Markup Language. It is a “tagged” form of ASCII text, in which the tags instruct the web browser how to display the referenced text, data, and images. For example, the HTML line

<b>This is an important statement</b>

presents the text string “**This is an important statement**” in a bold font. The <b> tag marks the start of the bold text, and the </b> marks the end of the bold text. There are many excellent books and free online tutorials that explain how to “mark up” text into an HTML file that will display a nicely formatted web page in your browser window.

The dynamic webserver is available at the standard TCP/IP port 80 used by web browsers. This port is a “well known port” that is assigned to HTTP (HyperText Transfer Protocol) data exchanges between a web browser (typically running on a PC) and the webserver (in this case running on the EtherSmart Wildcard). HTTP defines the request format that comes from the browser, and the response format that comes from the server. To learn about the HTTP 1.1 standard in all its detail, type “RFC 2616” into Google and read the referenced document.

The primary HTTP “method” command from the browser is the “GET” keyword, and this is the only HTTP method supported by the EtherSmart/WiFi webserver. The GET keyword is followed by a “URL”, or Uniform Resource Locator; it can also be referred to as a URI, or Uniform Resource Identifier. The URL is a web address on the target computer. For the purposes of this document, the URL does not include the target computer’s name, IP address or port. The URL starts with the / (forward slash) character and follows rules set out in the HTTP standard. A URL is a web page address as sent by the browser running on your desktop PC to the embedded web server. For the purposes of this document, the URL is the portion of the web address that is in the browser’s address bar after the IP address or computer name. For example, if you have assigned IP address 10.0.1.22 to the EtherSmart/WiFi Wildcard, and you type into your browser’s address bar:

10.0.1.22/index.html

then the URL as defined in this document is

/index.html

Each URL starts with a / character, and is case sensitive. Some URL’s include a “query field” that starts with the ? character and contains fieldname and value fields resulting from “forms” that are filled out by the user in the browser window. The software driver functions make it easy to extract data from these fields to direct the operation of the handler functions. In fact, form data from the browser provides an excellent way for the web interface to give commands to the embedded computer (to take data samples, extract data from memory and report the results to the browser, etc.)

The webserver contains a suite of functions that start with the **`HTTP_`** prefix to implement the dynamic webserver. These functions make it easy to respond to the browser request with a properly formatted “HTTP header” and content. The HTTP header specifies the content type (text/html, image/bitmap, etc.) and whether the referenced page can be “cached” by the browser for fast display after the initial loading. “Static” web pages can be safely cached, as they do not change over time. “Dynamic” web pages, on the other hand, should not be cached because they contain dynamic data that changes with time under the control of the application program.

You configure the webserver by defining and posting a “handler function” for each web address that is implemented. When the browser requests the web address URL, the webserver automatically executes the handler which performs any required actions and transmits the web page to the browser. As described in this section, powerful functions make it easy to use HTML “forms” to request specified data or stimulate particular actions from the Mosaic controller.

Because the web service is running in the Ethernet task, the web requests are handled automatically without the need for intervention by the application task. The Ethernet task's connection manager accepts the incoming request from the browser, identifies it as a web request by finding the "GET" keyword at the start of the first incoming line, matches the URL to those that have been declared, executes the corresponding handler to serve the requested web page, and closes the connection.

Your Browser Accesses the Dynamic Webserver

You'll use a web browser running on your PC to interact with the webserver running on the EtherSmart/WiFi Wildcard. As explained in the "Browser Notes" section above, popular browsers include Microsoft Internet Explorer, Netscape based browsers such as Firefox and Mozilla, and other high quality free browsers such as Opera. All of these browsers work with the web demonstration program that comes with the EtherSmart/WiFi Wildcard (see the demonstration program source code listings in the Appendices).

Additional considerations can limit the performance of some of these browsers if your application needs to serve out more complex web pages that require more than one TCP/IP connection per web page. This can occur, for example, when mixed text and image data originating from the Wildcard are served out in a single web page. The Lantronix hardware on the EtherSmart/WiFi Wildcard supports only one active connection at a time. However, the HTTP/1.1 standard (and consequently all browsers in their default configuration) expect the webserver to be able to host two simultaneous connections. A default-configured browser will try to open a second connection when two or more content types (for example, HTML text and a JPEG image) are present in a single web page. The second connection will typically be refused by the Lantronix hardware, causing an incomplete page load. The solution is to configure the browser to expect only one connection at a time from the webserver.

Appendix F explains how to reconfigure Internet Explorer to work with any web page that the EtherSmart or WiFi Wildcard can serve out. For the best solution, though, consider downloading the free Opera browser from www.opera.com and using it for all your interactions with the EtherSmart/WiFi Wildcard. It's free, the download file is compact, and the install takes only a few seconds. Once Opera is installed, simply go to its Tools menu, and select:

Preferences->Advanced->Network->Max Connections Per Server

and enter 1 in the box. Now you're ready to use Opera with the EtherSmart/WiFi Wildcard dynamic webserver.

Using the Image Converter to Create Web Resources

The most convenient way to create HTML or text web page strings as well as web images is to use the "Image Converter" program that is part of your Mosaic development environment. This program converts one or more files, each containing a single string, image or other resource, into a named 32-bit xaddress and count that can be used by your application program.

To define a set of text or text/HTML web page as resources that can be passed to the HTTP data output routines such as **`HTTP_Send_Buffer`** or **`HTTP_Send_2Buffers`**, simply place each string into a separate file with the file extension **`.str`** or **`.htm`** or **`.html`**. The file name and extension (with the dot replaced by an underscore) will become a part of the symbol name. All files to be converted should be in a single directory. Make sure you use C-compatible filenames that include only alphanumeric and _ (underscore) characters, and do not start with a numeral. Invoke the Image Converter from your development environment. In the Image Converter control panel, select the controller platform, and check the "Web files" box. If you are programming in Forth, click the "Advanced" menu and select Forth as the programming language.

If your web pages reference images that are to be served out by the EtherSmart/WiFi Wildcard itself, simply place files and the image files into the same directory. Each image file must have a valid file extension; the allowed file types are **`png`**, **`gif`**, or **`jpg`**. Note that bitmap (**`bmp`**) files are not convertible as web resources by the Image Converter, as bitmap images are reserved for the GUI (Graphical User Interface) toolkit which is also supported by the Image Converter.

In the "Directory" area of the Image Converter control panel, select the directory that contains the specified file(s). Then click on "Convert Files Now". A pair of files named **`image_data.txt`** and **`image_headers.h`** will be created by the Image Converter. The **`image_data.txt`** contains S-records and operating system commands that load the string image into flash memory on the controller. Because flash memory is nonvolatile, you only need to download this file once to the controller; it need not be reloaded until the web pages or images are edited. The **`image_headers.h`** file declares the xaddress and count (size) constants to be used by your application program.

For example, let's say that you have created a HTML "home page" and stored it in a file named **`home_body.html`** where the **`.html`** file extension indicates that the file contains HTML marked-up text. The **`image_headers.h`** file declares the following two macros with the appropriate numeric xaddress and count values:

```
#define HOME_BODY_HTML_XADDR 0x700360
#define HOME_BODY_HTML_SIZE 0x98E
```

Of course, the actual numeric xaddress value depends on the controller platform that you have selected and the other resources that are converted.

You can then use these macros as parameters passed to the **`HTTP_Send_Buffer`** or **`HTTP_Send_2Buffers`** functions to specify the email body string.

Loading Your Program

See Appendix A for a discussion of how to load the resources and the kernel extension library files onto the controller board before loading your custom application program.

Using the Dynamic Webserver

A comprehensive suite of pre-coded driver functions is available to simplify the implementation of an embedded webserver by your application program. Most of these functions begin with the

HTTP_ prefix to indicate that they are part of the web service suite. These can be classified as buffer management, HTTP header generation, data transfer, form processing, handler posting, and inter-task service management functions. Each of these is discussed in turn in the context of the constituent functions that implement the **Ether_Web_Demo** and **WiFi_Web_Demo** functions from the demo program. You can edit the definition of **main** in the demo source code file to call the version of the web demo that is appropriate for your Wildcard (EtherSmart or WiFi). Then running the web demo is as easy as typing **main** in the terminal window.

HTTP Buffer Management

Table 1-11 lists the web service buffer functions. The web service input and output buffers are called **HTTP_Inbuf** and **HTTP_Outbuf**. These can be used in any web handler function. The **Ether_Connection_Manager** routine that is running in the Ethernet task automatically stores the linefeed-delimited first line into the **HTTP_Inbuf** counted buffer, and examines the line to see if it starts with "GET ". If the first line starts with GET (which is the request keyword issued by a web browser), the connection manager invokes the web server.

Table 1-11 Web Service Buffer Management Functions.

Cat	**HTTP_Outbuf_Cat**
HTTP_Inbuf	**HTTP_Outbufsize**
HTTP_Inbufsize	**HTTP_OUTBUFSIZE_DEFAULT**
HTTP_INBUFSIZE_DEFAULT	**HTTP_Set_Inbuf**
HTTP_Outbuf	**HTTP_Set_Outbuf**

Web connections are typically handled "in the background" without the need for intervention by the application program. This is very convenient, but it implies that dedicated HTTP buffers must be used for web service. Hard to diagnose multitasking errors can result if the HTTP buffers are used to implement other services (such as tunneling or email), or if the **Ether_Inbuf** or **Ether_Outbuf** buffers are used to implement web handlers as well as tunneling or email services. In other words, <u>use separate buffers for web service as opposed to other Ethernet services</u>.

Each of the buffers **HTTP_Inbuf** and **HTTP_Outbuf** has a default maximum size that is set at initialization time. In this context, the "size" is the maximum number of data bytes (not including the count stored in the first 2 bytes of the buffer) that can fit in the buffer. The default sizes of the **HTTP_Inbuf** and **HTTP_Outbuf** are given by the constants **HTTP_INBUFSIZE_DEFAULT** (254 bytes) and **HTTP_OUTBUFSIZE_DEFAULT** (1022 bytes) respectively. **HTTP_Inbuf** can be smaller because it only needs to hold one input line at a time, while **HTTP_Outbuf** should be able to accommodate multi-line strings to serve out web page content. The functions **HTTP_Inbufsize** and **HTTP_Outbufsize** return the sizes of the **HTTP_Inbuf** and **HTTP_Outbuf**, respectively. The **Cat** and **HTTP_Outbuf_Cat** functions that write to the buffers accept the maximum buffer size as an input parameter, and ensure that they never write beyond the allowed buffer size.

If you need specify a new buffer base xaddress or size that is different from the values set at initialization time, use the functions **HTTP_Set_Inbuf** and **HTTP_Set_Outbuf**.

HTTP Header Generation

The HTTP standard specifies that a webserver should respond to a GET request with a properly formatted HTTP header followed by a single blank line followed by the content. This HTTP header should not be confused with the HTML **`<head>`** tag that specifies special content within an HTML page. The HTTP header tells the browser about the nature of the content that is about to be served by the embedded webserver. Table 1-12 lists the EtherSmart/WiFi Wildcard functions that make it easy to serve a valid HTTP header.

Table 1-12 HTTP Header Generation Functions.

HTTP_BINARY_DATA_CONTENT	**HTTP_Put_Content_Type**
HTTP_IMAGE_BITMAP_CONTENT	**HTTP_Put_Header**
HTTP_IMAGE_GIF_CONTENT	**HTTP_TEXT_HTML_CONTENT**
HTTP_IMAGE_JPEG_CONTENT	**HTTP_TEXT_PLAIN_CONTENT**
HTTP_IMAGE_PNG_CONTENT	

The HTTP header specifies the content type (text/html, image/bitmap, etc.) and whether the referenced page can be "cached" by the browser for fast display after the initial loading. "Static" web pages can be safely cached, as they do not change over time. "Dynamic" web pages, on the other hand, should not be cached because they contain dynamic data that changes with time under the control of the application program.

In the demo program, the **`Opera_Config_Page`** shown in Listing 1-9 serves out an unchanging HTML web page. The **`Home_Page`** function discussed below, on the other hand, serves HTML text plus the elapsed time between system initialization and the time the web page is served. Thus the static **`Opera_Config_Page`** handler should have a header that tells the browser that it is allowed to cache the page and restore it from memory if it is re-requested. The **`Home_Page`** function, on the other hand, must have a header that tells the browser not to cache the contents, as this would render the time report inaccurate.

The **`Opera_Config_Page`** source code in Listing 1-9 provides an example of how to serve out a simple static web page whose text is defined in a **`*.html`** resource file. Note that all web handler functions must accept a single input parameter (the modulenum) and return no parameters.

In this code excerpt, we first initialize some variables that are passed as parameters to **`Ether_Get_Line`**. One of the rules of calling operating system or kernel extension functions is that no function nesting is allowed: it is illegal to invoke one of these functions within the parameter list of another. By capturing the results returned by the **`HTTP_Outbuf`** and **`HTTP_Outbufsize`** functions into variables, we avoid the nesting prohibition when calling the EtherSmart driver functions invoked inside **`Opera_Config_Page`**.

Listing 1-9 A Handler Function from the Demo Program that Serves Static HTTP Header and HTML Text.

```
void Opera_Config_Page( int modulenum )
// this is the handler function for the /opera_configuration.html URL.
// sends the http header with static text/html content type,
// followed by opera_config.html text
{   xaddr http_outbuf_base = HTTP_Outbuf(modulenum);
```

```
    uint http_outbuf_size =  HTTP_Outbufsize(modulenum);
    HTTP_Put_Header(http_outbuf_base, http_outbuf_size);  // http header->outbuf
    HTTP_Put_Content_Type(http_outbuf_base, http_outbuf_size,0,
                          HTTP_TEXT_HTML_CONTENT);
                 // params: xlbuf,maxbufsize,dynamic?,content_type
                 // static text/html content ->outbuf;header is done
    HTTP_Send_LBuffer(http_outbuf_base, modulenum);
                 // send out header, ignore #bytes_sent
    HTTP_Send_Buffer(OPERA_CONFIG_HTML_XADDR, OPERA_CONFIG_HTML_SIZE, modulenum);
}                // send opera_config.html, ignore #bytes_sent
```

After initializing some local variables to the **HTTP_Outbuf** and **HTTP_Outbufsize**, there is a call to **HTTP_Put_Header**. This puts the following invariant portion of the EtherSmart/WiFi Wildcard's standard HTTP header into the specified LBuffer:

```
HTTP/1.1 200 OK
Server: Mosaic Industries Embedded Webserver
Connection: close
Content-Type:
```

There is a space after the Content-Type: field. The string count is stored in the first 2 bytes of the specified LBuffer, with the ASCII characters following. The first line announces the webserver as an HTTP/1.1 protocol server; the 200 OK means that the GET request has been received and a valid handler has been found that corresponds to the incoming URL. The second line announces the name of the webserver. The third line means that the webserver will close each connection after the GET request has been fulfilled, and "persistent connections" are not supported. The last line announces the content type which is typically filled in by **HTTP_Put_Content_Type**, but can also be accomplished using a **Cat** command with any arbitrary string to specify the content type. Note that one and only one empty line terminated by a carriage return/linefeed sequence must follow the header. The call to **HTTP_Put_Content_Type** does this in Listing 1-9. **HTTP_Put_Content_Type** expects as input parameters an LBuffer xaddress and maximum buffer size, a flag that is true if the web content is dynamic or false if it is static, and a constant that specifies the content type to be announced. As shown in Table 1-12, there are constants defined for a variety of data types, including text/html, text/plain, image/gif, image/jpeg, image/png, and binary data. The PNG image type is noteworthy because it is a standard non-proprietary image format.

In this example, the completed header placed in the **HTTP_Outbuf** by **HTTP_Put_Header** and **HTTP_Put_Content_Type** is:

```
HTTP/1.1 200 OK
Server: Mosaic Industries Embedded Webserver
Connection: close
Content-Type: text/html
```

This announces the page as text/html content. It is left to the browser to decide whether to cache the page; most browsers will cache web content by default.

If we had passed a true flag to the **HTTP_Put_Content_Type** to indicate that the web content was dynamic, the following header would have been produced:

```
HTTP/1.1 200 OK
Server: Mosaic Industries Embedded Webserver
Connection: close
Content-Type: text/html
Cache-Control: no-cache
```

The last line of this HTTP header for dynamic content tells the browser not to cache the content, thereby forcing the browser to reload the page each time it is requested. This ensures that the latest version of the dynamically generated web content is presented in the browser window.

Now that the HTTP header is stored in the **HTTP_Outbuf**, it is ready to be sent out to the web browser.

Rule for Web Handler Functions

All web handler functions must accept a single input parameter (the modulenum) and return no parameters.

HTTP Data Transfer Functions

The final two function calls in the **Opera_Config_Page** function in Listing 1-9 send out the HTTP header and the HTML web page content, respectively:

```
HTTP_Send_LBuffer(http_outbuf_base, modulenum);
                   // send out header, ignore #bytes_sent
HTTP_Send_Buffer(OPERA_CONFIG_HTML_XADDR, OPERA_CONFIG_HTML_SIZE, modulenum);
                  // send opera_config.html, ignore #bytes_sent
```

Recall that the local variable named **http_outbuf_base** is declared as an xaddress holding a 32-bit extended address, and is initialized to hold the value returned by **HTTP_Outbuf**. **modulenum** is the input parameter passed to the **Opera_Config_Page** function. **HTTP_Send_LBuffer** sends out the HTTP header which has been placed into the **HTTP_Outbuf**. Then **HTTP_Send_Buffer** sends out the HTML web page. The HTML source for this page can be found in the file named **opera_config.html** in the **\Resources** subdirectory of the EtherSmart/WiFi demo folder. The Image Converter program was used to convert this page and the other resources into S-record data that is loaded into flash when the **image_data.txt** file is downloaded to the Mosaic controller. The **image_headers.h** file contains the definitions of the 32-bit xaddress **OPERA_CONFIG_HTML_XADDR** constant and the 16-bit **OPERA_CONFIG_HTML_SIZE** constant that refer to the web page resource in memory.

Now we've reviewed all of the functions needed to send out the static web page from Listing 1-9. It's as easy as using **HTTP_Put_Header** and **HTTP_Put_Content_Type** to create the header, and calling **HTTP_Send_LBuffer** and **HTTP_Send_Buffer** to send the header and web content.

Table 1-13 lists the HTTP data transfer functions that should be invoked inside web service handlers to send out the HTTP header and web content. **HTTP_Send_2Buffers** sends two buffers one after another. **HTTP_Numbytes_Sent** returns the number of bytes sent by the last execution of **HTTP_Send_Buffer**, **HTTP_Send_LBuffer**, or **HTTP_Send_2Buffers**.

Table 1-13 HTTP Data Transfer Functions.

HTTP_Get_Timeout_Msec_Ptr	**HTTP_Send_Buffer**
HTTP_Numbytes_Sent	**HTTP_Send_LBuffer**

HTTP_Get_Timeout_Msec_Ptr	**HTTP_Send_Buffer**
HTTP_Send_2Buffers	**HTTP_Timeout_Msec_Ptr**

HTTP_Get_Timeout_Msec_Ptr returns the 32-bit xaddress of a 16-bit timeout variable used by the **Ether_Connection_Manager** during the attempt to identify an incoming web connection. The default value is 5000, corresponding to 5 seconds. If an incoming carriage-return/linefeed-delimited line is not available on the connection within the specified timeout, and if the contents of **HTTP_Enable_Ptr** are nonzero, the input operation will cease when the timeout is reached, and the HTTP identification will then proceed. Increasing the value of this timeout beyond its default value may improve the robustness of web service on some networks. Note, however, that an incoming serial tunneling connection that does not promptly send a carriage-return/linefeed-delimited line effectively delays the recognition of an incoming serial tunneling (as monitored by **Ether_Connect_Status**) by the value of this timeout parameter.

HTTP_Timeout_Msec_Ptr returns the xaddress that holds a 16-bit timeout for outgoing HTTP traffic in units of milliseconds. The maximum allowed timeout is 65,535. This timeout is used by **HTTP_Send_Buffer**, **HTTP_Send_LBuffer**, **HTTP_Send_2Buffers**, and the HTTP/GUI functions described in a later section. The default set by **Ether_Setup_Default** is 33000, corresponding to a 33 second timeout for outgoing HTTP traffic. You may need to increase this value if you are serving large files over slow or congested networks.

Note that you must use the paged store routine **StoreInt** to update the timeout parameters. For example, to change the HTTP incoming (Get) timeout to 3 seconds and the HTTP outgoing timeout to 40 seconds on module number 7, you could execute the following code:

```
xaddress http_get_timeout_msec_ptr = HTTP_Get_Timeout_Msec_Ptr(7);
xaddress http_timeout_msec_ptr = HTTP_Timeout_Msec_Ptr(7);
StoreInt(3000, http_get_timeout_msec_ptr);
StoreInt(40000, http_timeout_msec_ptr);
```

Web Handlers Are Posted to the Autoserve Array

The EtherSmart/WiFi embedded webserver is configured by defining and posting a "handler function" for each web address that is implemented. The **Opera_Config_Page** source code in Listing 1-9 provides an example of a handler function. Note that all web handler functions must accept a single input parameter (the modulenum) and return no parameters.

When the browser requests the web address URL (Uniform Resource Locator), the webserver looks for a match to the URL in the "Autoserve Array". If a match is found, the webserver running in the Ethernet task automatically executes the associated handler function which performs any required actions and transmits the web page to the browser. If a match is not found, the **HTTP_Default_Handler** is executed. This reports a "404 Page Not Found" error to the browser.

Table 1-14 lists the routines used to post web service handler functions. The "Autoserve Array" is a data structure that holds a pointer to each URL string, and the corresponding function pointer (execution address) of the handler associated with that URL. **HTTP_Autoserve_Ptr** returns an address that contains the 32-bit base address of this array. The Autoserve Array is set up with 32 rows by **Ether_Info_Init** and **WiFi_Info_Init** and their calling functions as listed in Table 1-3. **HTTP_AUTOSERVE_DEFAULT_ROWS** equals 32. This sets the maximum number of

URL/handler pairs that can be posted. If you need to post more handlers, use **HTTP_Is_Autoserve_Array** to declare and size a new version of the array; see its glossary entry for details. The functions **HTTP_Add_Handler** and **HTTP_Add_GUI_Handler** post the required information into this array, writing to the row pointed to by **HTTP_Index_Ptr**. Consequently, the programmer does not need to directly access the contents of this low-level data structure.

Table 1-14 HTTP Functions to Manage Posting of Web Page Handlers.

HTTP_Add_Handler	HTTP_Default_Handler_Ptr
HTTP_AUTOSERVE_DEFAULT_ROWS	HTTP_Index_Ptr
HTTP_Autoserve_Ptr	HTTP_Is_Autoserve_Array
HTTP_Default_Handler	

Listing 1-10 presents the **Web_Handler_Installation** function from the demo program to illustrate how web handlers are posted. Simply pass as parameters to the **HTTP_Add_Handler** function the base xaddress and length of the URL string, the handler function pointer (as a 32 bit xaddress), and the modulenum.

The first part of Listing 1-10 defines the URL strings for each web page handler in the demo program. Note that:

### all of the URLs start with the / (forward slash) character; and,

### it is good practice to always define a single forward slash string as a URL associated with the home page.

If a user types a bare IP address or computer name into the browser's address bar, the browser sends a single / character as the URL, and the correct response is typically to respond with the specified device's home page.

The next section of Listing 1-10 defines 32-bit function pointers for each of the handler functions. This is a bit tricky because the processor has a native 16-bit address space. While the Forth compiler always uses 32-bit paged xaddresses, the C compiler typically tracks addresses as 16-bit non-paged quantities. To obtain a 32-bit xaddress that contains both the page and the address of the handler function, we follow the procedure shown in Listing 1-10. First we use the **#ifdef** statement to find out which compiler we are using; the method shown works for the Fabius compiler which is used on all controllers using the V4.xx operating system. (The new PDQ line of products run the V6.xx operating system and the GCC tool chain, and use a different system to extract the 32-bit handler xaddress.) The **#pragma** and **#include </mosaic/gui_tk/to_large.h>** statements tell the compiler to go into a mode in which 32-bit function pointers can be generated and stored in non-volatile flash memory. Then the function pointers are defined. For example, the 32-bit variable

```
opera_config_page_ptr
```

is initialized to equal the 32-bit execution xaddress of the **Opera_Config_Page** routine defined in Listing 1-9 using the statement:

```
xaddr (*opera_config_page_ptr)(void)  = Opera_Config_Page;
```

After the nonvolatile 32-bit function pointers are captured, the **#pragma** and **to_large.h** compiler directives are reversed to return to normal compilation mode before compiling any functions.

Then the **Web_Handler_Installation** function is defined to post the handler functions. This function returns a nonzero error flag if the number of handlers posted exceeds the number of rows in the Autoserve Array (32 by default). We OR the error value returned by each invocation of **HTTP_Add_Handler** to produce the overall error result. The third call to **HTTP_Add_Handler** the **Web_Handler_Installation** function posts the handler for the **Opera_Config_Page** routine:

```
error |= HTTP_Add_Handler(STRING_XADDR(opera_url_str), strlen(opera_url_str),
            opera_config_page_ptr, modulenum);
```

The **STRING_XADDR** macro is defined in the **strdefs.h** file which is included at the top of the demo program. It converts the 16-bit string address to a 32-bit xaddress, inserting the page at the time the calling function runs. This macro is discussed in more detail in the "EtherSmart/WiFi Driver Data Structures: Passing String Extended Addresses as Function Parameters" section above. The **strlen** function is a standard C routine that returns the length of the specified string. The **opera_config_page_ptr** parameter is the 32-bit execution address (function pointer) to the handler routine. The modulenum completes the input parameter list in the call to **HTTP_Add_Handler** in Listing 1-10.

Listing 1-10 Posting the Web Handler Functions in the Demo Program.

```
// url strings:
char* slash_url_str = "/"; // synonym for home page url
char* index_url_str = "/index.html"; // home page url
char* opera_url_str = "/opera_configuration.html"; // opera page url
char* form_entry_url_str = "/form_entry.html"; // form entry page url
char* form_response_url_str = "/form_response.cgi"; // form response page url
char* logo_response_url_str = "/logo_response.png"; // logo response page url
char seconds_str[16];   // will hold elapsed seconds string

#ifdef      __FABIUS__
#pragma option init=.doubleword    // declare 32-bit function pointers in code area
#include </mosaic/gui_tk/to_large.h>
#endif   // __FABIUS__

xaddr (*home_page_ptr)(void)          = Home_Page;
xaddr (*opera_config_page_ptr)(void)  = Opera_Config_Page;
xaddr (*form_entry_page_ptr)(void)    = Form_Entry_Page;
xaddr (*form_response_page_ptr)(void) = Form_Response_Page;
xaddr (*logo_response_page_ptr)(void) = Logo_Response_Page;

#ifdef      __FABIUS__
#include </mosaic/gui_tk/fr_large.h>
#pragma option init=.init       // return the initialized variable area to RAM;
#endif   // __FABIUS__

int Web_Handler_Installation( int modulenum )
// call this after Ether_Task_Setup_Default
// point browser to raw ip or to ip/index.html to see the home web page.
// urls are case sensitive. any other url's serve out: page not found.
// returns nonzero error if too many handlers were added
// (limited by AUTOSERVE_DEFAULT_ROWS passed to ether_init)
{  int error = 0; // we'll OR error results together and return final result
   error |= HTTP_Add_Handler(STRING_XADDR(slash_url_str), strlen(slash_url_str),
            home_page_ptr, modulenum);
   error |= HTTP_Add_Handler(STRING_XADDR(index_url_str), strlen(index_url_str),
            home_page_ptr, modulenum);
```

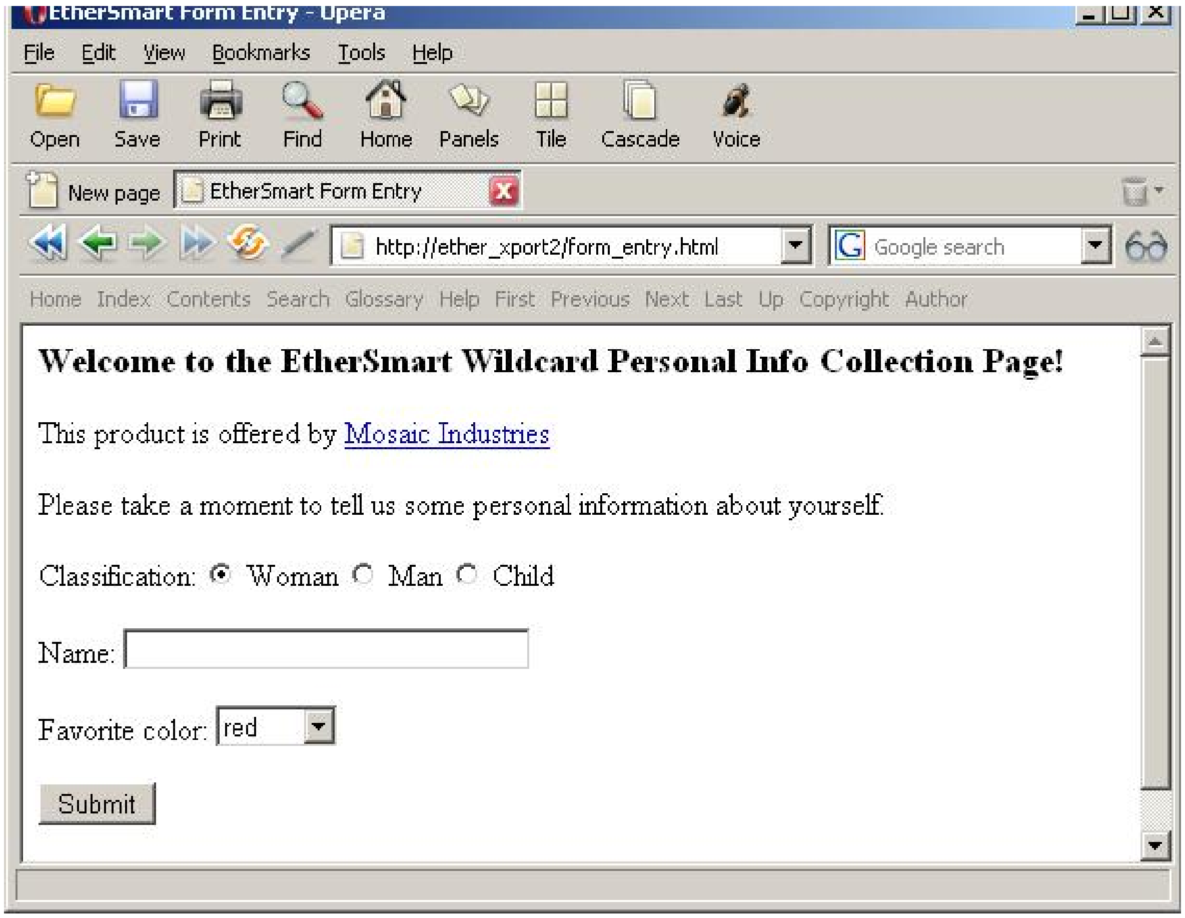
```
      error |= HTTP_Add_Handler(STRING_XADDR(opera_url_str), strlen(opera_url_str),
               opera_config_page_ptr, modulenum);
      error |= HTTP_Add_Handler(STRING_XADDR(form_entry_url_str),
                strlen(form_entry_url_str),
               form_entry_page_ptr, modulenum);
      error |= HTTP_Add_Handler(STRING_XADDR(form_response_url_str),
               strlen(form_response_url_str), form_response_page_ptr, modulenum);
      error |= HTTP_Add_Handler(STRING_XADDR(logo_response_url_str),
               strlen(logo_response_url_str), logo_response_page_ptr, modulenum);
      InitElapsedTime();  // start at zero so home page reports correct elapsed time
      return(error);
   }
```

Web Form Processing

It is easy to create "forms" in HTML. Forms provide a way for a user to enter data in text boxes. They allow a user to select items from drop-down menus, radio button lists, or check boxes that are presented in the web browser window. The user's selections are encoded as a set of "query fields" appended to the URL after a ? character. A set of pre-coded form processing functions in the EtherSmart/WiFi driver enable the application program to parse the query fields, extract the data, and take actions based on the results.

The demo program serves out a simple HTML form page as illustrated in Figure 1-12. This page is served out by the **`Form_Entry_Page`** handler function whose source code is shown in Listing 1-12. The form asks the user to select from radio buttons labeled "man", "woman" or "child", to type in their name, and to select their favorite color from a drop-down menu of colors "red", "blue", "yellow", or "green". Clicking the "Submit" button encodes the form results as a set of query fields appended to the **`form_response.cgi`** URL. This URL is handled by the **`Form_Response_Page`** function in Listing 1-12. The **`.cgi`** file extension on the response URL refers to the "Common Gateway Interface" which deals with query fields. Note that the file names and extensions chosen for the URL are arbitrary (you can choose anything reasonable that you like), but it is a good idea to select file names and extensions that indicate the nature of the web service content or action.

While the form shown in Figure 1-12 is somewhat whimsical, it is meant to be suggestive of very useful capabilities. For example, your application could serve out a web form that asks the user to select a set of digital output channels and the corresponding values to which they should be set (high or low). The form could ask the user to specify a set of analog data acquisition channels, and the application program could respond by printing a table of the latest values for the specified channels. In other words, forms provide a means to instruct the application program to take specified actions, as well as a means to request specified data.


EtherSmart Form Entry - Opera
File Edit View Bookmarks Tools Help
Open Save Print Find Home Panels Tile Cascade Voice
New page
EtherSmart Form Entry
http://ether_xport2/form_entry.html
Google search
Home Index Contents Search Glossary Help First Previous Next Last Up Copyright Author

Welcome to the EtherSmart Wildcard Personal Info Collection Page!

This product is offered by Mosaic Industries

Please take a moment to tell us some personal information about yourself.

Classification: [✓ Woman] Man Child

Name:

Favorite color: red

Submit

Figure 1-12 Form entry web page served out by the EtherSmart/WiFi Demo Program.

Listing 1-11 presents the HTML source text that generates the web page in Figure 1-12. This page is compiled as a resource string by the image converter, resulting in constant definitions **FORM_PAGE_HTML_XADDR** and **FORM_PAGE_HTML_SIZE** that refer to the string. These constants are used in the **Form_Entry_Page** handler function that serves out the web page as shown in Listing 1-12.

Listing 1-11 Contents of the form_page.html Resource File.

```
<html>
<head><title>EtherSmart Form Entry</title></head>
<body>
<H3>Welcome to the EtherSmart Wildcard Personal Info Collection Page!</H3><p>
This product is offered by <a href="http://www.mosaic-industries.com">Mosaic
Industries</a>
<p>Please take a moment to tell us some personal information about yourself.<p>
<form method=GET action="/form_response.cgi">
Classification:
<input type=radio checked name=classification value="woman"> Woman
<input type=radio name=classification value="man"> Man
<input type=radio name=classification value="child"> Child
<p>Name: <input type=text name=name_id size=32 maxlength=80>
<p>Favorite color:
```

```
<select name=color size=1>
<option>red
<option>blue
<option>yellow
<option>green
</select>
<p><input type=submit>
</form>
<p> <a href="/index.html">Go back home</a><p></body></html>
```

Listing 1-11 provides an example of how to write the HTML code that generates a form web page. The **`<html>`** tag starts the file, and is balanced by the **`</html>`** tag at the end of the file. These tags tell the browser that the enclosed text is HTML. The **`<head>`** and **`</head>`** tags declare the HTML header area of the file. In the HTML header, the title of the web page is declared as "EtherSmart Form Entry". It is good practice to declare an informative title for each web page, as many browsers display the title.

The **`<body>`** tag starts the displayed body of the web page in Listing 1-11. The **`<H3>`** and **`</H3>`** tags declare the start and end of a heading style for the welcome message. The **`<p>`** tag means "start a new displayed paragraph" and results in the display of a blank line in the browser window; note that there is no **`</p>`** tag in HTML.

The link to the Mosaic Industries web site is generated by the HTML **`<a>`** "anchor" tag, and the **`href`** keyword specifies the link destination:

```
<a href="http://www.mosaic-industries.com">Mosaic Industries</a>
```

The start of the form portion of the page that is to be completed and submitted by the user is declared using the HTML code:

```
<form method=GET action="/form_response.cgi">
```

The "GET" method is the only method that is supported by the EtherSmart/WiFi Wildcard. The **`action`** keyword declares the URL of the routine that responds to the form request. This URL invokes the **`Form_Response_Page`** function whose source code is presented in Listing 1-12.

The first part of the form declares a set of radio buttons. A radio button set allows only one of the choices to be active; use check boxes if you want the user to be able to select multiple items from a single set of options. Each radio button declaration is of the form:

```
<input type=radio name=classification value="man"> Man
```

The **`name`** keyword declares the fieldname, and the **`value`** keyword specifies the value. The browser passes the fieldname followed by an = character followed by the selected value as part of the query field in the URL. Note that the text outside the <input > tag boundaries is simply HTML text that is displayed in the form. There is no requirement that the value field declared in the tag be identical to the displayed text. In this example, the value field and the displayed text differ only in their capitalization.

The next portion of the form declares a textbox using the line:

```
<p>Name: <input type=text name=name_id size=32 maxlength=80>
```

This declares a textbox 32 characters wide, with a maximum number of 80 characters accepted; the textbox scrolls if the number of characters exceeds 32. The fieldname is **`name_id`**, and the field value will equal the contents typed in by the user. There are some subtleties, though. The HTTP standard requires that certain characters may not be included in a URL and must be "escaped" by encoding the character as **`%hexhex`**, where **`hexhex`** represents the two hexadecimal characters that specify the ASCII value of the desired character. In addition, the browser converts each space character (ASCII 0x20) into the + character. The following is a minimal list of characters that are escaped by the browser:

```
; / ? : @ = & < > " # % { } | \ ^ ~ [ ] `
```

To make it easy to deal with this situation, the EtherSmart/WiFi driver contains the **`HTTP_Unescape`** and **`HTTP_Plus_To_Space`** functions that make it easy to recover the original text as typed by the user.

The next part of the form in Listing 1-11 is a drop-down menu that is started with the tag:

```
<select name=color size=1>
```

The **`name`** keyword defines the fieldname, and the **`size`** keyword specifies how many items are visible at one time to the user. The optional **`multiple`** keyword, if included in the select tag, allows the user to select more than one option.

The form is ended with the tags:

```
<input type=submit></form>
```

When the user clicks the "Submit" button, the selected choices are encoded as query fields appended to the URL. For example, if the user selects "man" as the classification, types "Randy Newman!" in the name box, and chooses "yellow" as the favorite color, the following URL is sent by the browser to the EtherSmart/WiFi Wildcard:

```
/form_response.cgi?classification=man&name_id=Randy+Newman%21&color=yellow
```

This URL illustrates how the query fields are formatted by the browser. The query fields start with the **`?`** character, and each subsequent field is separated by the **`&`** delimiter. Within each field is the fieldname followed by the **`=`** character, followed by the value field. Note that the exclamation point after the name is escaped as %21, and the space between "Randy" and "Newman" is converted to a plus sign in the URL. The **`Form_Response_Page`** function calls **`HTTP_Unescape`** and **`HTTP_Plus_To_Space`** to process the URL query fields and recover the original text as typed by the user. This URL invokes the **`Form_Response_Page`** function in Listing 1-12 which parses the query fields following the **`?`** character in the URL, and serves out a response page. For the URL shown above, the response presented in the web browser looks like this:

> **Thanks for your response!**
>
> It must be great to be a man with a name like Randy Newman! .
>
> It's good to hear that yellow is your favorite color!

Table 1-15 presents the pre-coded EtherSmart/WiFi form processing functions that are used to parse and respond to the query fields in the URL. **`HTTP_Parse_URL`** is automatically called by the webserver and is typically not directly called by the programmer. It sets up the initial parsing pointers to point to the first query field (if present) in the URL. **HTTP_Parse_URL** initializes the **HTTP_URL_Ptr** to return the xaddress of the start of the URL in the **HTTP_Inbuf**. It sets the **HTTP_URL_Base_Count** to equal the number of characters before the start of the query field, and sets the **HTTP_URL_Full_Count** to the total number of characters in the URL. If there is no query field, then the **HTTP_URL_Base_Count** equals the **HTTP_URL_Full_Count**.

The **HTTP_Fieldname_Ptr** and **HTTP_Fieldname_Count** return the xaddress and count, respectively, of the next fieldname. In the example URL, the first fieldname is **classification** and its count is 14 (the number of characters in **classification**). The **HTTP_Value_Ptr** and **HTTP_Value_Count** return the xaddress and count, respectively, of the next value which appears after the **=** character. In the example URL above, the first value is **man** and its count is 3 (the number of characters in **man**).

Table 1-15 HTTP Form Processing Functions.

HTTP_Fieldname_Count	HTTP_Unescape
HTTP_Fieldname_Ptr	HTTP_URL_Base_Count
HTTP_Parse_URL	HTTP_URL_Full_Count
HTTP_Plus_To_Space	HTTP_URL_Ptr
HTTP_Status_Ptr	HTTP_Value_Count
HTTP_To_Next_Field	HTTP_Value_Ptr

To see an example of how these form processing functions work, we can take a close look at the source code of the **Form_Response_Page** function in Listing 1-12. As usual, we first define and load a set of local variables to avoid nesting of operating system function calls. These local variables capture the starting values of the return values for the **HTTP_Outbuf**, **HTTP_Outbufsize**, **HTTP_Value_Ptr**, and **HTTP_Value_Count** for the specified modulenum. The standard procedure for HTTP header generation is followed, invoking **HTTP_Put_Header**, **HTTP_Put_Content_Type**, and **HTTP_Send_LBuffer** to send the HTTP header, declaring the content as static HTML text. Then **HTTP_Send_Buffer** is used to send the **form_response_str**, with the **STRING_XADDR** macro used in the standard fashion to convert the string base address to a 32-bit xaddress, with **strlen** providing the string count. At this point, the **Form_Response_Page** function has served out the HTTP header and displayed the string:

> **Thanks for your response!**
>
> It must be great to be a

Next, **Store_Int** is invoked by **Form_Response_Page** to zero the 16-bit count at the start of the **HTTP_Outbuf**, thereby starting a new string in the LBuffer. If the first value was entered by the user (that is, if it has a non-zero count), the value is appended to the **HTTP_Outbuf**. Otherwise, the neutral classification string “person” is added to the buffer using **HTTP_Outbuf_Cat**. This completes the processing of the man/woman/child part of the form.

HTTP_To_Next_Field is called by **Form_Response_Page** to advance to the **name_id** field associated with the textbox, and the return values of **HTTP_Value_Ptr** and **HTTP_Value_Count** are stored in the local variables.

Then, to avoid confusing the parsing routines when **`HTTP_Unescape`** is called, **`HTTP_To_Next_Field`** is called again to advance to the color field. Then the **`HTTP_Value_Ptr`** and **`HTTP_Value_Count`** return values are stored in the **`next_http_value_ptr`** and **`next_http_value_count`** variables. This technique protects us against a situation in which the user types any of the query field delimiters such as **`&`** or **`=`** as part of the name entered into the textbox. When we unescape the textbox string, these delimiters would appear in the URL and confuse the parsing routines. We "look ahead" to avoid this problem.

If the user entered a name in the textbox, **`Form_Response_Page`** calls **`HTTP_Plus_To_Space`** and **`HTTP_Unescape`** to convert the text in the URL back into the characters that the user typed. In the example URL above, these steps deal with the **`+`** between "Randy" and "Newman", and recover the **`!`** character that the user typed. Note that we capture the count returned by **`HTTP_Unescape`**, as this routine shortens the textbox substring by two characters for each escape sequence that it processes. If the user did not enter anything in the textbox, then we add the neutral term "yours" to the **`HTTP_Outbuf`** using **`HTTP_Outbuf_Cat`**.

The next few lines in the **`Form_Response_Page`** function add the favorite color string to the **`HTTP_Outbuf`** using **`HTTP_Outbuf_Cat`**. A line of text containing the ending tags **`</body></html>`** is added to the **`HTTP_Outbuf`**, and the contents of **`HTTP_Outbuf`** are sent to the browser using **`HTTP_Send_LBuffer`**. This completes the form response, and the browser displays a message that summarizes the form entries that were made.

While this example is purely text-based in its response, keep in mind that the same coding techniques can be applied to instruct the Mosaic controller to perform sophisticated data acquisition, control, and status reporting in response to the inputs received via a web form.

Listing 1-12 Form Processing Web Request and Response Pages.

```
// form_response page strings:

char* form_response_str = "<html><head><title>Form Response</title></head><body>\r\n\
<H3>Thanks for your response!</H3><p><p>It must be great to be a ";

char* person_str = "person";
char* name_announce_str = " with a name like ";
char* yours_str = "yours";
char* good_to_hear_str = ". It's good to hear that ";
char* rainbow_str = "rainbow";
char* favorite_str = " is your favorite color!</body></html>";
// note: in form_page_str, we mark one of the radio buttons as checked; otherwise,
// the browser can skip the field entirely,
// then we would have to match the fieldnames!

void Form_Entry_Page( int modulenum )
// responds to url= /form_entry.html
// places FORM_TEXT into the buffer as a longstring after the http header.
// we send directly from compiled string in flash to avoid copy into ether_out buffer;
// this method is ideal for serving static text.
{  xaddr http_outbuf_base = HTTP_Outbuf(modulenum);
   uint http_outbuf_size =  HTTP_Outbufsize(modulenum);
   HTTP_Put_Header(http_outbuf_base, http_outbuf_size);  // http header->outbuf
   HTTP_Put_Content_Type(http_outbuf_base, http_outbuf_size,0,
                         HTTP_TEXT_HTML_CONTENT);
                   // params: xlbuf,maxbufsize,dynamic?,content_type
                   // static text/html content ->outbuf;header is done
   HTTP_Send_LBuffer(http_outbuf_base, modulenum);
                  // send out header, ignore #bytes_sent
   HTTP_Send_Buffer(FORM_PAGE_HTML_XADDR, FORM_PAGE_HTML_SIZE, modulenum);
}                  // send form_page.html, ignore #bytes_sent


void Form_Response_Page ( int modulenum )
// this is the handler function that responds to url= /form_response.cgi
// incoming request from browser looks like this:
// GET /form_response.cgi?classification=<man/woman/child>&name_id=<namestring>
// &color=<red/blue/yellow/green>
// We respond:
// It's great to be a <man/woman/child/person>
//   with a name like <namestring/yours>.
// It's good to hear that <red/blue/yellow/green/rainbow> is your favorite color.
{  xaddr http_outbuf_base = HTTP_Outbuf(modulenum);
   uint http_outbuf_size =  HTTP_Outbufsize(modulenum);
   xaddr http_value_ptr = HTTP_Value_Ptr(modulenum); // load value,cnt for 1st field
   uint http_value_count =  HTTP_Value_Count(modulenum);
   xaddr next_http_value_ptr; // used for look-ahead field to avoid unescape problems
   uint next_http_value_count;// used for look-ahead field to avoid unescape problems
   HTTP_Put_Header(http_outbuf_base, http_outbuf_size);  // http header->outbuf
   HTTP_Put_Content_Type(http_outbuf_base, http_outbuf_size,0,
                      HTTP_TEXT_HTML_CONTENT);
                  // params: xlbuf,maxbufsize,dynamic?,content_type
                  // static text/html content ->outbuf;header is done
   HTTP_Send_LBuffer(http_outbuf_base, modulenum);
                  // send out header, ignore #bytes_sent
   HTTP_Send_Buffer(STRING_XADDR(form_response_str), strlen(form_response_str),
                      modulenum);   // send form_response_str, ignore #bytes_sent
   StoreInt(0, http_outbuf_base);      // reset lcount to start fresh in buffer
   if(http_value_count) // if first field was entered, add it to response
      HTTP_Outbuf_Cat(http_value_ptr, http_value_count, modulenum);
   else              // else add "person" to response
      HTTP_Outbuf_Cat(STRING_XADDR(person_str), strlen(person_str), modulenum);
   HTTP_Outbuf_Cat(STRING_XADDR(name_announce_str), strlen(name_announce_str),
                  modulenum);
                     // add " with a name like " to response
   HTTP_To_Next_Field(modulenum); // go to name_id field, ignore #chars_advanced
   http_value_ptr = HTTP_Value_Ptr(modulenum);  // load value,cnt for name_id field
   http_value_count =  HTTP_Value_Count(modulenum);
   HTTP_To_Next_Field(modulenum); // go to color field, ignore #chars_advanced
   next_http_value_ptr = HTTP_Value_Ptr(modulenum);// load value,cnt for color field
   next_http_value_count =  HTTP_Value_Count(modulenum);
   if(http_value_count) // if a name was entered, process it and add it to response
   {  HTTP_Plus_To_Space(http_value_ptr, http_value_count); // get rid of + in name
      http_value_count = HTTP_Unescape(http_value_ptr, http_value_count);
      HTTP_Outbuf_Cat(http_value_ptr, http_value_count, modulenum);
   }
   else              // else add "yours" to response
      HTTP_Outbuf_Cat(STRING_XADDR(yours_str), strlen(yours_str), modulenum);
   HTTP_Outbuf_Cat(STRING_XADDR(good_to_hear_str), strlen(good_to_hear_str),
                  modulenum);
                     // add ". It's good to hear that " to response
   if(next_http_value_count)  // if color field was entered, add it to response
      HTTP_Outbuf_Cat(next_http_value_ptr, next_http_value_count, modulenum);
   else              // else add "rainbow" to response
      HTTP_Outbuf_Cat(STRING_XADDR(rainbow_str), strlen(rainbow_str), modulenum);
   HTTP_Outbuf_Cat(STRING_XADDR(favorite_str), strlen(favorite_str), modulenum);
                     // add " is your favorite color!</body></html>" to response
   HTTP_Send_LBuffer( http_outbuf_base, modulenum);
}                    // send response, ignore #bytes_sent
```

HTTP Inter-Task Service Management Functions

The service management functions that run the webserver are the same as those described in the section titled “Serial Tunneling Inter-Task Service Management Functions”; they are listed in Table 1-10. Unlike the other services that have been discussed such as serial tunneling and email, the web service is performed autonomously by the Ethernet task. After the handlers are posted to the Autoserve Array, non-GUI web service requests are handled by the Ethernet task with no intervention by the application task. The only exception is the implementation of a GUI (Graphical User Interface) “remote front panel” feature which must be synchronized to the application task as discussed in a later section.

An Example of a Dynamic Web Page with a Remote Image Reference

The EtherSmart home page (served out by both the EtherSmart and WiFi Wildcards) is shown in Figure 1-13 as it appears in a browser. It presents some welcoming and explanatory text, provides a link to the form page discussed in the prior section, and invites the user to link to a stand-alone image as discussed in the next section. The home page presents a link to the Mosaic Industries website at www-mosaic-industries.com, and displays an image that is hosted by the Mosaic site. It invites the user to link to the Opera browser website at www.opera.com, and links to an informative page that describes the advantages of Opera and tells how to configure it for use with the EtherSmart/WiFi Wildcard. It recommends the use of the Putty TCP/IP terminal program for serial tunneling and interactive TCP/IP communications with the EtherSmart/WiFi Wildcard. Finally, it displays a dynamically generated elapsed time since system initialization.

It is easy to implement these features using HTML and a bit of software. Listing 1-13 shows the HTML source code that generates the home page. This HTML file is processed as a resource by the Image Converter program as explained above and in Appendix A. Many of the HTML tags in this file have already been discussed in earlier sections, including the following tags:

```
<html> </html>  <head> </head>  <title> </title>  <H3> </H3>  <p>
```

The link to the form entry page is coded as follows:

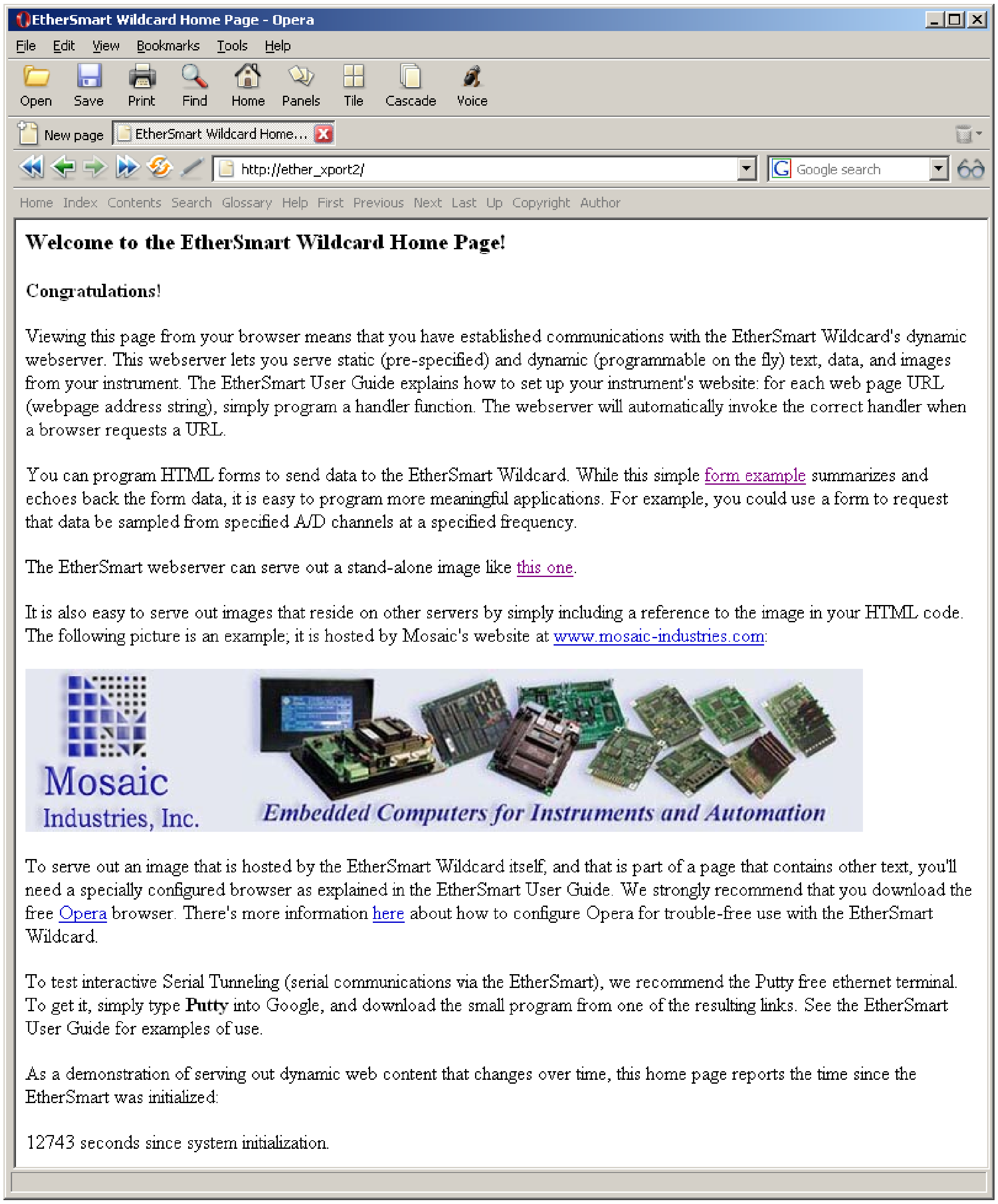
```
While this simple <a href="/form_entry.html">form example</a> summarizes
```

This uses the standard `<a> </a>` “anchor” tags, with the `href` keyword specifying the target URL, and the remaining text inside the tag specifying the link text displayed by the browser (“form example” in this case).

A similar link to the stand-alone logo image is served out by the following HTML:

```
The EtherSmart webserver can serve out a stand-alone image like
<a href="/logo_response.png">this one</a>.
```

This anchor invokes the `/logo_response.png` URL; recall that PNG is a convenient public domain image compression format. The `Logo_Response_Page` function that is associated with this URL is described in the next section.


EtherSmart Wildcard Home Page - Opera
File Edit View Bookmarks Tools Help
Open Save Print Find Home Panels Tile Cascade Voice
New page
EtherSmart Wildcard Home...
http://ether_xport2/
Google search
Home Index Contents Search Glossary Help First Previous Next Last Up Copyright Author
Mosaic Industries, Inc.
Embedded Computers for Instruments and Automation

Welcome to the EtherSmart Wildcard Home Page!

Congratulations!

Viewing this page from your browser means that you have established communications with the EtherSmart Wildcard's dynamic webserver. This webserver lets you serve static (pre-specified) and dynamic (programmable on the fly) text, data, and images from your instrument. The EtherSmart User Guide explains how to set up your instrument's website: for each web page URL (webpage address string), simply program a handler function. The webserver will automatically invoke the correct handler when a browser requests a URL.

You can program HTML forms to send data to the EtherSmart Wildcard. While this simple form example summarizes and echoes back the form data, it is easy to program more meaningful applications. For example, you could use a form to request that data be sampled from specified A/D channels at a specified frequency.

The EtherSmart webserver can serve out a stand-alone image like this one.

It is also easy to serve out images that reside on other servers by simply including a reference to the image in your HTML code. The following picture is an example; it is hosted by Mosaic's website at www.mosaic-industries.com:

To serve out an image that is hosted by the EtherSmart Wildcard itself, and that is part of a page that contains other text, you'll need a specially configured browser as explained in the EtherSmart User Guide. We strongly recommend that you download the free Opera browser. There's more information here about how to configure Opera for trouble-free use with the EtherSmart Wildcard.

To test interactive Serial Tunneling (serial communications via the EtherSmart), we recommend the Putty free ethernet terminal. To get it, simply type **Putty** into Google, and download the small program from one of the resulting links. See the EtherSmart User Guide for examples of use.

As a demonstration of serving out dynamic web content that changes over time, this home page reports the time since the EtherSmart was initialized:

12743 seconds since system initialization.

Figure 1-13 Home page served out by the EtherSmart/WiFi Wildcard as seen in the browser window.

The following HTML presents a link to the Mosaic Industries website, and displays a picture:

```
It is also easy to serve out images that reside on other servers by simply
including a reference to the image in your HTML code.  The following picture
is an example; it is hosted by Mosaic's website at
<a href="http://www.mosaic-industries.com">www.mosaic-industries.com</a>:
```

```
<p><img src="http://www.mosaic-industries.com/ESmart_Images/webbanner_logo1.jpg">
```

The link to the Mosaic Industries website uses the standard anchor tag syntax as described above. The **`<img>`** tag refers to the image called **`webbanner_logo1.jpg`** which resides on the Mosaic Industries website. While the home page text in Figure 1-13 image is served directly by the EtherSmart/WiFi Wildcard, the displayed image actually comes from another webserver. You can use this technique to serve large, colorful images without burdening the embedded controller itself. This technique uses the power of the Internet to enhance the visual appeal of web pages served by the EtherSmart/WiFi Wildcard.

Listing 1-13 Contents of home_body.html Resource File.

```
<html><head><title>EtherSmart Wildcard Home Page</title></head><body>
<H3>Welcome to the EtherSmart Wildcard Home Page!</H3><p>
<H4>Congratulations!</H4><p>
Viewing this page from your browser means that you have established
communications with the EtherSmart Wildcard's dynamic webserver.
This webserver lets you serve static (pre-specified) and
dynamic (programmable on the fly) text, data, and images from your instrument.
The EtherSmart User Guide explains how to set up your instrument's website:
for each web page URL (webpage address string), simply program a handler function.
The webserver will automatically invoke the correct handler
when a browser requests a URL.
<p>
You can program HTML forms to send data to the EtherSmart Wildcard.
While this simple <a href="/form_entry.html">form example</a> summarizes
and echoes back the form data, it is easy to program more meaningful applications.
For example, you could use a form to request that data be sampled from specified
A/D channels at a specified frequency.
<p>
The EtherSmart webserver can serve out a stand-alone image like
<a href="/logo_response.png">this one</a>.
<p>
It is also easy to serve out images that reside on other servers by simply
including a reference to the image in your HTML code.  The following picture
is an example; it is hosted by Mosaic's website at
<a href="http://www.mosaic-industries.com">www.mosaic-industries.com</a>:
<p><img src="http://www.mosaic-industries.com/ESmart_Images/webbanner_logo1.jpg">
<p>
To serve out an image that is hosted by the EtherSmart Wildcard itself,
and that is part of a page that contains other text, you'll need
a specially configured browser as explained in the EtherSmart User Guide.
We strongly recommend that you download
the free <a href="http://www.opera.com">Opera</a> browser.
There's more information <a href="/opera_configuration.html">here</a>
about how to configure Opera for trouble-free use with the EtherSmart Wildcard.
<p>
To test interactive Serial Tunneling (serial communications via the EtherSmart),
we recommend the Putty free ethernet terminal.  To get it, simply type <b>Putty</b>
into Google, and download the small program from one of the resulting links.
See the EtherSmart User Guide for examples of use.
<p>
As a demonstration of serving out dynamic web content that changes over time,
this home page reports the time since the EtherSmart was initialized:<p>
```

Listing 1-14 presents the source code that serves out the EtherSmart home page. Most of the code follows the procedures outlined in earlier sections, capturing the buffer xaddress and count into local variables, serving the HTTP header using **`HTTP_Put_Header`**, **`HTTP_Put_Content_Type`** and **`HTTP_Send_LBuffer`**. The unique aspect of this web page is that it inserts dynamically generated data, reporting the elapsed time since system initialization. The **ReadElapsedSeconds** function returns a 32-bit time count that is captured into the **elapsed_sec** variable. Then the standard **sprintf** C function is invoked in the argument list of **HTTP_Outbuf_Cat** to convert the time to a string and append it to the **HTTP_Outbuf** buffer. **HTTP_Send_LBuffer** serves out the dynamically generated elapsed time to the browser.

This is a simple example of dynamically generated web content. Using these techniques, your application can dynamically display the status of your instrument to a remote operator via the web.

Listing 1-14 Dynamic Web Page Source Code.

```
void Home_Page( int modulenum )
// this is the handler function for the /index.html and the / URLs.
// sends the http header with dynamic (because of changing time stamp) text/html
// content type,  followed by home_body_str text and the elapsed seconds since the
// system was initialized.
{  ulong elapsed_sec;
   xaddr http_outbuf_base = HTTP_Outbuf(modulenum);
   uint http_outbuf_size =  HTTP_Outbufsize(modulenum);
   HTTP_Put_Header(http_outbuf_base, http_outbuf_size);  // http header->outbuf
   HTTP_Put_Content_Type(http_outbuf_base, http_outbuf_size, 1,
                      HTTP_TEXT_HTML_CONTENT);
               // params: xlbuf,maxbufsize,dynamic?,content_type
               // dynamic text/html content ->outbuf;header is done
   HTTP_Send_LBuffer(http_outbuf_base, modulenum);
              // send out header, ignore #bytes_sent
   HTTP_Send_Buffer(HOME_BODY_HTML_XADDR, HOME_BODY_HTML_SIZE, modulenum);
               // send home_body.html, ignore #bytes_sent
   StoreInt(0, http_outbuf_base);      // reset lcount to start fresh in buffer
   elapsed_sec = ReadElapsedSeconds();  // get elapsed time, used by sprintf...
   HTTP_Outbuf_Cat((xaddr) seconds_str,
              (int) sprintf(seconds_str,"%ld",elapsed_sec),modulenum);
        // params: xstring_to_add,cnt_to_add,module
        // convert seconds to string, add line & crlf to longstring, update lcnt
   HTTP_Outbuf_Cat(STRING_XADDR(time_announce_str), strlen(time_announce_str),
              modulenum);
   // add to lbuffer: " seconds since system initialization.</body></html>"
   HTTP_Send_LBuffer( http_outbuf_base, modulenum);
}  // send elapsed time, ignore #bytes_sent
```

Serving Out a Stand-Alone Image

The home page served out by the EtherSmart/WiFi Wildcard provides a link to the **/logo_response.png** URL, a stand-alone image of the Mosaic Logo. The logo image is in a file named **mosaic_logo.png**, and the Image Converter program converts it into a resource specified by the constants **MOSAIC_LOGO_PNG_XADDR** and **MOSAIC_LOGO_PNG_SIZE**. The handler for this URL is the **Logo_Reponse_Page** function shown in Listing 1-15. Because the web page contains only one type of data (in this case, a PNG image), it can be properly rendered by any browser in the browser's default configuration. The response function shown in Listing 1-15 is simple. After capturing the **HTTP_Outbuf** and **HTTP_Outbufsize** return results in local variables, **HTTP_Put_Header** and **HTTP_Put_Content_Type** are called to create the header. The content type is specified by the constant **HTTP_IMAGE_PNG_CONTENT**, and the image is declared as static because it does not change over time. The size of the HTTP header is retrieved out of the first 16 bits of the **HTTP_Outbuf** by the **Fetch_Int** function; recall that because the buffer is in paged memory, the paged fetch operator must be used. Then the convenient **HTTP_Send_2Buffers** function is invoked to send first the HTTP header and then the logo image. Note that the starting xaddress of the HTTP header is specified by `http_outbuf_base+2`, as the first 2 bytes of this LBuffer contain the number of bytes stored. The `MOSAIC_LOGO_PNG_XADDR` and `MOSAIC_LOGO_PNG_SIZE` constants specify the logo image data that comprises the content of the served page.

Listing 1-15 Handler Function that Serves Out an Image.

```
void Logo_Response_Page ( int modulenum )
// this is the handler function that responds to url= /logo_response.png
// assumes that logo.s2 has been loaded into memory. serves it as static image.
{  xaddr http_outbuf_base = HTTP_Outbuf(modulenum);
   uint http_outbuf_size =  HTTP_Outbufsize(modulenum);
   int http_header_size;
   HTTP_Put_Header(http_outbuf_base, http_outbuf_size);  // http header->outbuf
   HTTP_Put_Content_Type(http_outbuf_base, http_outbuf_size,0,
                      HTTP_IMAGE_PNG_CONTENT);
                  // params: xlbuf,maxbufsize,dynamic?,content_type
                  // static png image type->outbuf, cnt in first 2bytes; header's done
   http_header_size = FetchInt(http_outbuf_base);  // get size of http header
   HTTP_Send_2Buffers(http_outbuf_base+2, http_header_size,
                  MOSAIC_LOGO_PNG_XADDR, MOSAIC_LOGO_PNG_SIZE, modulenum);
                  // params: xbuf1,cnt1,xbuf2,count2,modulenum
}                 // send header&image, ignore numbytes_sent
```

Initializing the Demonstration Program

Listing 1-16 shows the high level startup functions of the Demo program. `Ether_Web_Demo` initializes the EtherSmart Wildcard, starts the Ethernet task, and runs the web demo. `WiFi_Web_Demo` initializes the WiFi Wildcard, starts the Ethernet task, and runs the web demo. These two functions are very similar: the former invokes `Ether_Task_Setup_Default`, while the latter invokes `WiFi_Task_Setup_Default`. `main` can be edited to call either `Ether_Web_Demo` or `WiFi_Web_Demo`. Each web demo function invokes `Web_Handler_Installation` which was discussed in the Section titled "Web Handlers Are Posted to the Autoserve Array".

The `main` function can be edited to call either `Ether_Web_Demo` or `WiFi_Web_Demo`. You can run the demonstration program by compiling it and loading it as described in Appendix A, and then typing `main` at the QED Terminal.

As described in earlier sections, you can interactively type the following commands after executing `main`; all work for both EtherSmart and WiFi Wildcards:

```
Ether_Task_Setup_Default(  )  // full initialization
Ether_Set_Defaults(  )  // sets mosaic factory defaults; returns error code
Ether_Set_Local_IP(  int my_ip1, int my_ip2, int my_ip3,  int my_ip4 )  // sets IP
Ether_IP_Report(  )   // summarizes the IP address, gateway address, and netmask.
Ether_Ping(  int remote_ip1, int ip2, int ip3,  int ip4)  // prints ping report
Ether_Monitor_Demo( int modulenum )  // runs interactive monitor on modulenum
Tunnel_Test(  )  // waits for incoming connection via Putty or other Ether terminal
Email_Test(   )  // sends a test email; you must edit function's source code first
```

Recall that when interactively typing these commands, there is no space between the function name and the `(` character, and there must be at least one space after the `(` character. All of these functions have been described in the text of this document.

Listing 1-16 High Level Initialization Functions in the Demo Program.

```
void Ether_Web_Demo( void )
// for EtherSmart Wildcard: point browser to raw ip or to
//  ip/index.html to see the home web page.  urls are case sensitive.
// any other url's serve out: page not found.
// returns nonzero error if too many handlers were added
// (limited by AUTOSERVE_DEFAULT_ROWS passed to ether_init)
{  Ether_Task_Setup_Default();
   Web_Handler_Installation(E_MODULENUM);       // ignore too_many_handlers? flag
}
void WiFi_Web_Demo( void )
// for WiFi Wildcard: point browser to raw ip or to
//  ip/index.html to see the home web page.  urls are case sensitive.
// any other url's serve out: page not found.
// returns nonzero error if too many handlers were added
// (limited by AUTOSERVE_DEFAULT_ROWS passed to ether_init)
{  WiFi_Task_Setup_Default();
   Web_Handler_Installation(E_MODULENUM);       // ignore too_many_handlers? flag
}
void main( void )
{         // comment in the correct version for your wildcard:
    Ether_Web_Demo();
//  WiFi_Web_Demo();
}
```

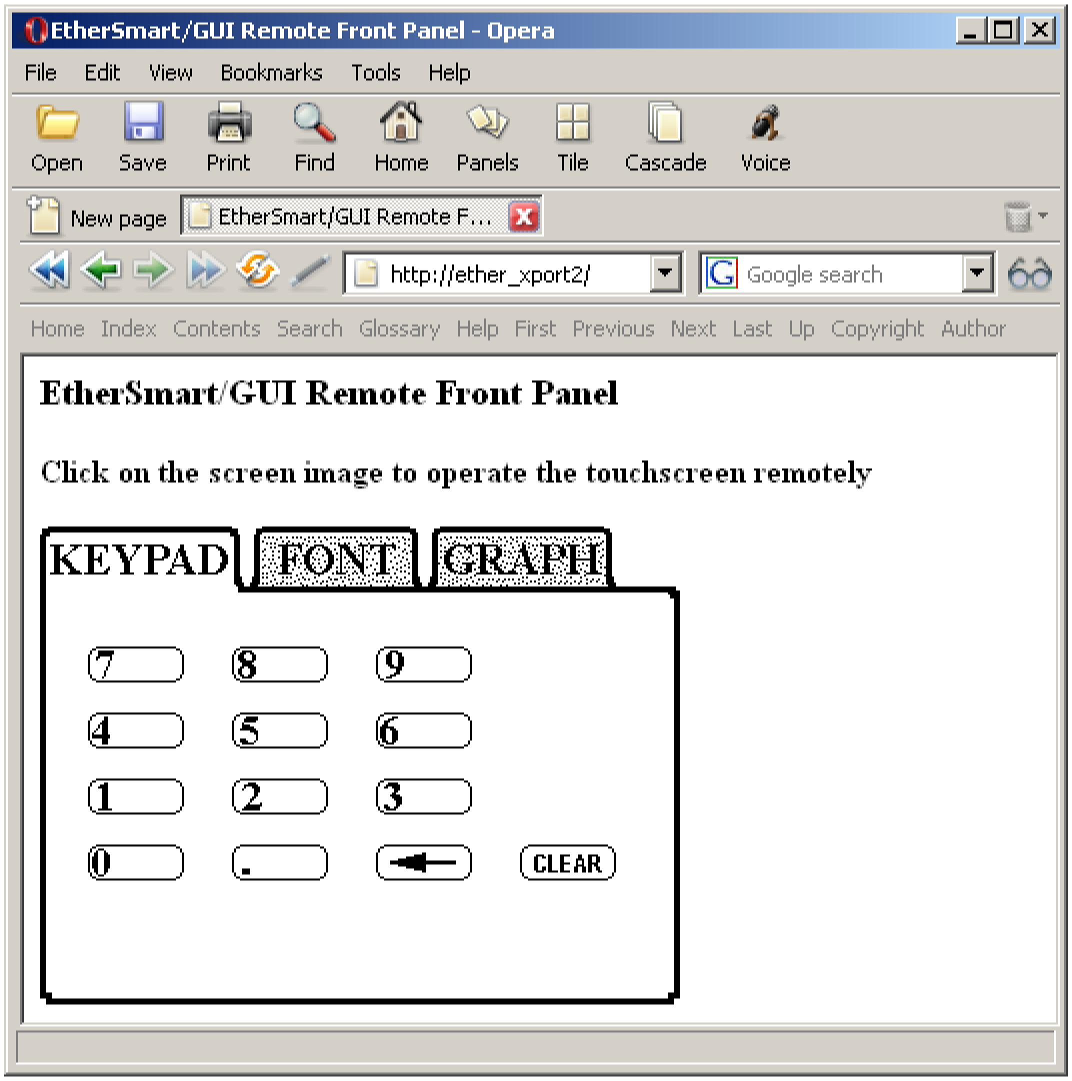
Implementing a "Remote Front Panel" Using the Webserver

Some Mosaic controllers offer a touchscreen-based "Graphical User Interface" (GUI) that enables a user to control the instrument by touching buttons and menus on the screen. The software that runs the GUI is called the "GUI Toolkit" and is available as a pre-coded "kernel extension" library from Mosaic Industries. These GUI-based controllers include the QVGA Controller and the PDQScreen; the feature is not recommended for use on the QScreen due to hardware limitations that slow its ability to serve the screen image over the web.

An EtherSmart or WiFi Wildcard can be added to a touchscreen-based controller to implement a web-based "Remote Front Panel" that allows a user to operate the unit via a web browser. The browser displays a bitmapped image of the GUI screen exactly as it appears on the controller's LCD (Liquid Crystal Display) as illustrated in Figure 1-14. Clicking on the web-based image with a mouse has the same effect as touching a button on the touchscreen: both the local screen image and the web image are updated. There is a delay before the new image appears on the web, because it takes time to extract, format and serve out the bitmapped image. Use of the remote front panel feature can expand the capabilities of an instrument, offering the possibility of web-based remote control, reporting, and diagnostics via an Ethernet or WiFi connection.

To implement this feature in your application, you can incorporate the source code presented in Appendix C (for C programmers) or E (for Forth programmers). The GUI Toolkit functions that enable this feature are described in a special section at the end of the EtherSmart/WiFi Wildcard Glossary document.

The remainder of this section describes how the remote front panel works.


EtherSmart/GUI Remote Front Panel - Opera
File Edit View Bookmarks Tools Help
Open Save Print Find Home Panels Tile Cascade Voice
New page
EtherSmart/GUI Remote F...
http://ether_xport2/
Google search
Home Index Contents Search Glossary Help First Previous Next Last Up Copyright Author
KEYPAD
FONT
GRAPH
7
8
9
4
5
6
1
2
3
0
.
CLEAR

EtherSmart/GUI Remote Front Panel

Click on the screen image to operate the touchscreen remotely

Figure 1-14 Web-based remote front panel running on the QVGA Controller.

The HTML text shown in Listing 1-17 creates the image shown in Figure 1-14:

Listing 1-17 Remote front panel HTML source from gui_response_text.html.

```
<html><head><title>EtherSmart/GUI Remote Front Panel</title></head>
<body>
<H3>EtherSmart/GUI Remote Front Panel</H3><p>
<H4>Click on the screen image to operate the touchscreen remotely</H3><p><p>
<a href="/gui_response.cgi"><img src="/screen_image.bmp" ismap></a>
</body></html>
```

This file is made into a resource by the Image Converter program as described in the section above titled “Introduction to the Dynamic Webserver: Using the Image Converter to Create Web Resources”, and in Appendix A. Briefly, we place the `gui_response_text.html` file along with the GUI demo program’s button graphics in a directory, invoke the Image Converter from the development environment, select the correct platform, check both “GUI images” and “Web Files”, and click “Convert Files Now”. The HTML in Listing 1-17 can now be referred to by the generated constants **GUI_RESPONSE_TEXT_HTML_XADDR** and **GUI_RESPONSE_TEXT_HTML_SIZE**; the constant names are based on the specified file name.

A standard set of **<html>**, **<head>**, and **<title>** tags start the web page generation in Listing 1-17. The **<H3>** header style is used to display the “EtherSmart/GUI Remote Front Panel” announcement. The **<H4>** tag formats the instruction to “Click on the screen image to operate the touchscreen remotely”. The next line is the key to the technique:

```
<a href="/gui_response.cgi"><img src="/screen_image.bmp" ismap></a>
```

The **ismap** keyword at the end of the tag tells the browser that the image should be treated as an “imagemap”. Any mouse click in the image causes the browser to send the x and y coordinates of the mouse click (relative to the upper left corner of the image) as a specially formatted query field appended to the URL. The EtherSmart/WiFi driver’s pre-coded **HTTP_Imagemap** function extracts the coordinates from the URL for use by the GUI Toolkit. The anchor tag declares an **href** reference to the URL **/gui_response.cgi** that is invoked each time the image is clicked. The handler for this URL invokes the **HTTP_Imagemap** function to extract the x and y mouse click coordinates, and passes them to the GUI Toolkit’s **Simulated_Touch_To_Image** function. This processes the coordinates and updates the screen as if an actual touch to the front panel had occurred. The **<img>** tag specifies that the image itself is served out by the **/screen_image.bmp** URL as specified by the image **src** keyword. The **</body>** and **</html>** tags complete the page source.

To start the web display, we need a starting web handler function that serves the initial screen image when the webserver is first connected to the controller. This web handler is called **Remote_Panel_Start** in Listing 1-18 and is designated as the home page invoked by either the **/** or **/index.html** URL. This function creates an HTTP header in the **HTTP_Outbuf** using the **HTTP_Put_Header** and **HTTP_Put_Content_Type** functions. It then calls the **Init_Screen_Image** function, which invokes the GUI Toolkit function **Screen_To_Image** to place the bitmap image in a buffer.

Because this web handler calls a GUI Toolkit function (**Screen_To_Image**), task synchrony issues come into play. The GUI Toolkit often runs in its own task, and the webserver runs in the Ethernet task. To avoid multitasking errors and resource conflicts, functions that involve both GUI and HTTP functions must be synchronized with one another. This is done automatically if some simple programming rules are followed:

Programming rules for the Ethernet/GUI Remote Front Panel on the EtherSmart/WiFi Wildcard

1. Each web handler function that invokes a GUI Toolkit function (**Graphic_To_Image**, **Screen_To_Image**, **Simulated_Touch_To_Image**, or **Screen_Has_Changed**) must be posted using **HTTP_Add_GUI_Handler**. The standard non-GUI function **HTTP_Add_Handler** must not be used for these handler functions.
2. Each web handler function that invokes a GUI Toolkit function must use one and only one call to **HTTP_GUI_Send_2Buffers** to send out the HTTP header and content.
3. Each GUI handler function that changes the screen appearance must call the **Screen_Has_Changed** function.
4. For QScreen Controllers only, the initialization statement **Globalize_TVars(TVARS);** must be invoked when the application starts up.

Following these rules allows the various tasks to work together in harmony to implement the remote front panel.

Let's continue examining how the functions in Listing 1-18 work. We now have the initial screen image as shown in Figure 1-14 in the web browser window as placed by the **Remote_Panel_Start** handler function. Now the user clicks on one of the keypad buttons (say, the "7" button) in the image. This causes the following URL and formatted "imagemap" query string to be sent to the EtherSmart/WiFi Wildcard:

```
/gui_response.cgi?43,69
```

This URL is handled by the **GUI_Response** handler function whose source is shown in Listing 1-18. This handler calls the **HTTP_Imagemap** function which extracts the x and y values from the URL (43 and 69, respectively in this example). Because C functions can return only one parameter, the two 16-bit coordinates are combined into a 32-bit value that is declared as a **TWO_INTS** union as defined in the **types.h** file in the mosaic include directory of the C distribution. The handler stores x and y into separate variables, and passes them to the GUI Toolkit function **Simulated_Touch_To_Image**. This function treats the coordinates as those of an actual touch on the front panel, calls any GUI handlers associated with the button, updates the local display, and stores a bitmap image of the display into the specified buffer at **BITMAP_BUFFER**. **GUI_Response** then invokes **HTTP_GUI_Send_2Buffers** to serve out the HTTP header from **HTTP_Outbuf**, and the HTTP content from the resource string specified by **GUI_RESPONSE_TEXT_HTML_XADDR** and **GUI_RESPONSE_TEXT_HTML_SIZE**.

The web content served out by the **GUI_Response** handler function is shown in Listing 1-17. It uses the **img** tag to refer to the URL **/screen_image.bmp.** This URL is serviced by the **Screen_Image_Response** handler in Listing 1-18. When this handler is called, the bitmap image of the screen has already been created and is sitting in the **BITMAP_BUFFER**. The first 4 bytes at **BITMAP_BUFFER** contain a 32-bit count, with the data following. After generating the HTTP header using the standard function calls, the **Screen_Image_Response** function gets the size of the bitmap by invoking

```
image_size = FetchInt(BITMAP_BUFFER+2)
```

The remote front panel feature is limited to image sizes that can be represented as a 16-bit number, so the 2 most significant bytes of the count can be skipped. A call to **HTTP_Send_2Buffers** serves out the HTTP header from **HTTP_Outbuf**, and serves the bitmap image data starting at **BITMAP_BUFFER+4**. Note that the **Screen_Image_Response** handler does not call any GUI Toolkit routines, so it need not follow the first two rules for Ethernet/GUI functions (shown in the box, above). Thus it can use **HTTP_Send_2Buffers** instead of **HTTP_GUI_Send_2Buffers**, and it is posted via the **HTTP_Add_Handler** function in **Install_GUI_Web_Handlers** as shown in Listing 1-18.

Some additional steps are required to complete the remote front panel. These are additions to the GUI demo program that creates and manages the touchscreen-based image.

### Place a call to the **Ether_Check_GUI** function in the application task loop that runs the GUI toolkit. The GUI toolkit function is called **Service_GUI_Events** on the QVGA controller and **Service_Touch** on the QScreen. Search for the GUI Toolkit routine (or its equivalent on other controllers) in the GUI demo program, and insert the following function call in the main task loop:

```
Ether_Check_GUI( E_MODULENUM ); // service remote front panel handler
```

### Add a call to the **Screen_Has_Changed** in each GUI handler function that changes the screen appearance.

Please read the "Notes" at the bottom of Listing 1-18 for a reminder of these and other required steps.

The GUI demo program which implements the touchscreen display screen shown in Figure 1-14 is started by the **GUI_Demo** function. To start both the GUI demo program and the remote front panel feature, we invoke the high level function named **Ether_GUI_Web_Demo** (for an EtherSmart Wildcard) or **WiFi_GUI_Web_Demo** (for a WiFi Wildcard) in Listing 1-18. The definition of **main** can be edited to invoke the function that corresponds to your Wildcard (EtherSmart or WiFi). It calls **Ether_Task_Setup_Default**, **Install_GUI_Web_Handlers**, and **GUI_Demo** to initialize the required tasks and services. After calling this high-level initialization function, browsing into the EtherSmart/WiFi Wildcard's home page (port 80 at the Wildcard's IP address) displays the remote front panel in your browser.

For C programmers, the **C_Ether_gui_demo_includer.txt** file in the **EtherSmart_WC_Code\Ether_GUI_Demo** directory loads the **install.txt** and **image_data.txt** files and the C compiler output file for the Ethernet GUI demo program. For Forth programmers, the **Ether_gui_demo_includer.4th** file in the **\Ether_Demo** directory loads these files and the source code file for the Ethernet/WiFi demo program. These commented files provide a useful template to manage the loading of this demonstration program.

Listing 1-18 Excerpts from Ethernet/WiFi/GUI remote front panel source code.

```
// ********************* URL STRINGS **************************
char* slash_url_str = "/"; // synonym for home page url
char* index_url_str = "/index.html"; // home page url
char* screen_image_str = "/screen_image.bmp"; // screen image url
char* gui_response_str = "/gui_response.cgi"; // gui response url

// *********************** WEBSERVER ***************************
// Notes: this works very nicely with Opera, the recommended browser.

// the following is created using the Image Converter program, creates 2 constants:
// GUI_RESPONSE_TEXT_HTML_XADDR
// GUI_RESPONSE_TEXT_HTML_SIZE

void Screen_Image_Response( int modulenum )
// handler for /screen_image.bmp url. serves the screen image.
// assumes that Simulated_Touch_To_Image has run.
// xbuffer contains 32-bit count (NOTE!) followed by image data.
// marks as no-cache (always reloads).
// NOTE: this function is NOT a gui handler; it should be posted by http_add_handler.
{  xaddr http_outbuf_base = HTTP_Outbuf(modulenum);
   uint http_outbuf_size =  HTTP_Outbufsize(modulenum);
   uint image_size = FetchInt((xaddr) (BITMAP_BUFFER+2));   // get size of image
   uint http_header_size;
   HTTP_Put_Header(http_outbuf_base, http_outbuf_size);  // http header->outbuf
   HTTP_Put_Content_Type(http_outbuf_base, http_outbuf_size, 1,
            HTTP_IMAGE_BITMAP_CONTENT);
                  // params: xlbuf,maxbufsize,dynamic?,content_type
                  // dynamic bitmap image content ->outbuf;header is done
   http_header_size = FetchInt(http_outbuf_base);  // get size of http header
   HTTP_Send_2Buffers(http_outbuf_base+2, http_header_size,
      (xaddr) (BITMAP_BUFFER+4), image_size, modulenum);
}                 // send http header and bitmap image

void GUI_Response( int modulenum )
// clickable ismap version of the screen image that runs gui toolkit simulated touch.
// url = /gui_response.cgi
// this routine is called via ether_check_gui which runs in the application task
// that runs Service_GUI_Events.  it is posted using http_gui_add_handler.
{  xaddr http_outbuf_base = HTTP_Outbuf(modulenum);
   uint http_outbuf_size =  HTTP_Outbufsize(modulenum);
   uint http_header_size, x, y;
   TWO_INTS x_and_y;
   x_and_y = HTTP_Imagemap( modulenum); // get mouseclick x,y coordinates
   x = x_and_y.twoNums.msInt; // get x from most significant 16bits
   y = x_and_y.twoNums.lsInt; // get y from least significant 16bits
   Simulated_Touch_To_Image(BITMAP_BUFFER,BITMAP_MAXBYTES,BITMAP_SCREEN_FORMAT,x,y);
   HTTP_Put_Header(http_outbuf_base, http_outbuf_size);  // http header->outbuf
   HTTP_Put_Content_Type(http_outbuf_base, http_outbuf_size, 0,
           HTTP_TEXT_HTML_CONTENT);
                  // params: xlbuf,maxbufsize,dynamic?,content_type
                  // dynamic bitmap image content ->outbuf;header is done
   http_header_size = FetchInt(http_outbuf_base);  // get size of http header
   HTTP_GUI_Send_2Buffers(http_outbuf_base+2, http_header_size,
      GUI_RESPONSE_TEXT_HTML_XADDR, GUI_RESPONSE_TEXT_HTML_SIZE, modulenum);
}                 // send http header and content

_Q int Init_Screen_Image( void )
// if return value=0, image has been created
{  return Screen_To_Image((xaddr) BITMAP_BUFFER, BITMAP_MAXBYTES,
            BITMAP_SCREEN_FORMAT);
}


void  Remote_Panel_Start( int modulenum )
// serves an initial clickable ismap version of the screen image.
// url = /index.html
// puts initial version of screen into bitmap_buffer.
// this routine is called via ether_check_gui which runs in the application task
// that runs Service_GUI_Events.  it is posted using http_gui_add_handler
{  xaddr http_outbuf_base = HTTP_Outbuf(modulenum);
   uint http_outbuf_size =  HTTP_Outbufsize(modulenum);
   int http_header_size;
   HTTP_Put_Header(http_outbuf_base, http_outbuf_size);  // http header->outbuf
   HTTP_Put_Content_Type(http_outbuf_base, http_outbuf_size,0,
           HTTP_TEXT_HTML_CONTENT);
                  // params: xlbuf,maxbufsize,dynamic?,content_type
                  // static html type->outbuf, cnt in first 2bytes; header's done
   Init_Screen_Image(); // ignore return value; image has been created
   http_header_size = FetchInt(http_outbuf_base);  // get size of http header
   HTTP_GUI_Send_2Buffers(http_outbuf_base+2, http_header_size,
            GUI_RESPONSE_TEXT_HTML_XADDR, GUI_RESPONSE_TEXT_HTML_SIZE, modulenum);
                  // params: xbuf1,cnt1,xbuf2,count2,modulenum
}                 // send header&content, ignore numbytes_sent

#ifdef      __FABIUS__
#pragma option init=.doubleword    // declare 32-bit function pointers in code area
#include </mosaic/gui_tk/to_large.h>
#endif   // __FABIUS__

xaddr (*screen_image_response_ptr)(void) = Screen_Image_Response;
xaddr (*gui_response_ptr)(void)          = GUI_Response;
xaddr (*remote_panel_start_ptr)(void)    = Remote_Panel_Start; // home page

#ifdef      __FABIUS__
#include </mosaic/gui_tk/fr_large.h>
#pragma option init=.init       // return the initialized variable area to RAM;
#endif   // __FABIUS__

int Install_GUI_Web_Handlers( int modulenum )
// call this after ETHER_Task_Setup_DEFAULT  point browser to raw ip or to
//  ip/index.html to see the home GUI web page = remote front panel.
// urls are case sensitive.
// any other url's serve out: page not found.
// returns nonzero error if too many handlers were added
// (limited by AUTOSERVE_DEFAULT_ROWS passed to ether_init)
// Some of the gui web handlers in this example are handled by the application task.
{  int error = 0; // we'll OR error results together and return final result
   error |= HTTP_Add_GUI_Handler(STRING_XADDR(slash_url_str), strlen(slash_url_str),
            remote_panel_start_ptr, modulenum);
   error |= HTTP_Add_GUI_Handler(STRING_XADDR(index_url_str), strlen(index_url_str),
            remote_panel_start_ptr, modulenum);
   error |= HTTP_Add_Handler(STRING_XADDR(screen_image_str), strlen(screen_image_str),
            screen_image_response_ptr, modulenum);
   error |= HTTP_Add_GUI_Handler(STRING_XADDR(gui_response_str),
            strlen(gui_response_str), gui_response_ptr, modulenum);
   InitElapsedTime();      // start at zero so home page reports correct elapsed time
   return(error);
}

_Q void Ether_GUI_Web_Demo( void )
{  Ether_Task_Setup_Default();                       // init EtherSmart WC, start
ethernet task
Install_GUI_Web_Handlers(E_MODULENUM);  // setup web; ignore error flag
GUI_Demo();       // start the gui demo on local touchscreen
}                   // NOTE: for QScreen, GUI_Demo MUST call Globalize_TVars(TVARS);

_Q void WiFi_GUI_Web_Demo( void )
{  WiFi_Task_Setup_Default();                        // init WiFi WC, start ethernet
task
Install_GUI_Web_Handlers(E_MODULENUM);  // setup web; ignore error flag
GUI_Demo();       // start the gui demo on local touchscreen
}                   // NOTE: for QScreen, GUI_Demo MUST call Globalize_TVars(TVARS);

void main( void )
```

```
{                                       // comment in the correct version for your
wildcard:
Ether_GUI_Web_Demo();
// WiFi_GUI_Web_Demo();
}

// NOTES: place a call to Ether_Check_GUI(E_MODULENUM); in the application task
// that invokes the GUI Routine
// (for example, after the call to Service_GUI_Events on a QVGA Controller system).
// Add a call to Screen_Has_Changed();
//  in each GUI handler that changes screen appearance.
// QScreen Controllers require a call to Globalize_TVars(TVARS); in the init function.
// When building tasks, do NOT include the statement NEXT_TASK = TASKBASE;
//  because we want to allow multiple tasks to be built.
// Type WARM or COLD before re-running the program.
```

Appendix A: Installing the Software

The EtherSmart/WiFi Wildcard device driver software is provided as a pre-coded modular runtime library, sometimes known as a "kernel extension" because it enhances the on-board kernel's capabilities. The GUI Toolkit that runs the touchscreen-based Graphical User Interface is also provided as a kernel extension library. One or more kernel extension libraries can be generated simultaneously. The library functions are accessible from C and Forth.

Generating the EtherSmart/WiFi Library

Mosaic Industries can provide you with a web site link that will enable you to create a packaged kernel extension that has drivers for all of the hardware that you have on your system. In this way the software drivers are customized to your needs, and you can generate whatever combination of drivers you need. Make sure to choose the correct controller platform and specify the EtherSmart/WiFi Wildcard Driver in the list of kernel extensions you want to generate, and download the resulting `packages.zip` file to your hard drive.

For convenience, a separate pre-generated kernel extension for the EtherSmart/WiFi Wildcard is available from Mosaic Industries on the software distribution CD. Look in the directory named:

`C:\Mosaic\Demos_and_Drivers\EtherSmart_WC_Code\Ether_Demo\Library`

The `V4_4` kernel subdirectory contains the EtherSmart kernel extension for QCard, QScreen, and Mosaic Handheld products. The `V4_09` kernel subdirectory is for the QVGA controller and other products based on the QED-Board.

The kernel extension is downloaded as a "zipped" file named `packages.zip`. Unzipping it (using, for example, winzip or pkzip) extracts the following files:

###	`readme.txt`	Provides summary documentation about the library.
###	`install.txt`	Installation file, to be loaded to COLD-started Mosaic Controller.
###	`library.4th`	Forth name headers and utilities; #include in top of Forth program.
###	`library.c`	C callers for all functions in library; #include in C code.
###	`library.h`	C prototypes for all functions; #include in extra C files.

The `library.c` and `library.h` files are only needed if you are programming in C. The `library.4th` file is only needed if you are programming in Forth. The uses of all of these files are explained below.

We recommend that you move the relevant files to the same directory that contains your application source code, or into a convenient nearby sub-directory.

Creating Web Page and Image Resources with the Image Converter

The most convenient way to create HTML or text web page strings as well as web images is to use the "Image Converter" program that is part of your Mosaic development environment. This program converts one or more files, each containing a single string, image or other resource into a named 32-bit xaddress and count that can be used by your application program.

To define a set of text or text/HTML web page as resources that can be passed to the HTTP data output routines such as `HTTP_Send_Buffer` or `HTTP_Send_2Buffers`, simply place each string into a separate file with the file extension `.str` or `.htm` or `.html` and a file name that will become a part of the symbol name. All files to be converted should be in a single directory. Make sure you use C-compatible filenames that include only alphanumeric and _ (underscore) characters, and do not start with a numeral. Invoke the Image Converter from your development environment. In the Image Converter control panel, select the controller platform, and check the "Web files" box. If you are programming in Forth, click the "Advanced" menu and select Forth as the programming language.

If your web pages reference images that are to be served out by the Wildcard itself, simply place the files containing these images into the same directory. Each image file must have a valid file extension; the allowed file types are `png`, `gif`, or `jpg`. Note that bitmap (`bmp`) files are not convertible as web resources by the Image Converter, as bitmap images are reserved for the GUI (Graphical User Interface) toolkit which is also supported by the Image Converter.

In the "Directory" area of the control panel, select the directory that contains the specified file(s). Then click on "Convert Files Now". A pair of files named `image_data.txt` and `image_headers.h` will be created by the Image Converter. The `image_data.txt` contains S-records and operating system commands that load the string image into flash memory on the controller. Because flash memory is nonvolatile, you only need to download this file once to the controller; it need not be reloaded until you change the resources. The `image_headers.h` file declares the xaddress and count (size) constants to be used by your application program.

For example, let's say that you have created a rather lengthy email body and stored it in a file named `mail_body.html` where the `.html` file extension indicates that the file contains HTML text. The `image_headers.h` file declares the following two macros with the appropriate numeric xaddress and count values:

```
#define HOME_BODY_HTML_XADDR 0x700360
#define HOME_BODY_HTML_SIZE 0x98E
```

Of course, the actual numeric xaddress value depends on the controller platform that you have selected and the other resources that are converted.

You can then use these macros as parameters passed to the `HTTP_Send_Buffer` or `HTTP_Send_2Buffers` functions to specify the string.

Loading the Software onto the Controller

To use the kernel extension, the runtime kernel extension code contained in the **`install.txt`** file must first be loaded into the flash memory of the Mosaic Controller. Power up the Mosaic Controller, connect it to the serial port, and start the Mosaic Terminal software. If you have not yet tested your Mosaic Controller and terminal software, please refer to the documentation provided with your Mosaic Controller. Once you can hit enter and see the 'ok' prompt returned in the terminal window, type

```
COLD
```

to ensure that the board is ready to accept the kernel extension install file. Use the "Send File" menu item of the terminal to download the **`install.txt`** to the Mosaic Controller.

If you have created resources such as web pages or images using the image converter file, use the "Send File" menu item of the terminal to download the **`image_data.txt`** file to the Mosaic Controller.

Now, type

```
COLD
```

again and the kernel has been extended with the driver and resources! Once **`install.txt`** has been loaded, it does not need to be reloaded each time you revise your source code. Similarly, once **`image_data.txt`** has been loaded, it does not need to be reloaded unless and until you decide to change the web page or image sources.

For C programmers, the **`C_Ether_WiFi_demo_includer.txt`** file in the **`Ether_WiFi_WC_Code\Ether_WiFi_Demo`** directory loads the **`install.txt`** and **`image_data.txt`** files and the C compiler output file for the Ethernet demo program. For Forth programmers, the **`Ether_WiFi_demo_includer.4th`** file in the **`Ether_WiFi_WC_Code\Ether_WiFi _Demo`** directory loads these files and the source code file for the Ethernet/WiFi demo program.

As explained in the next section, only the much smaller **`library.c`** and **`library.h`** (for C programmers) or **`library.4th`** file (for Forth programmers) need to be included with each compile/load cycle during programming.

Using the EtherSmart/WiFi Driver with C

If the **`library.c`** and **`library.h`** files are in the same directory as your other C source code files, use the following directive in your source code file:

```
#include "library.c"
```

If the files are in a different directory, place the path before the filename separated by the **`\`** character to specify where the file is in relation to the source code file. The **`library.c`** file contains calling primitives that implement the functions in the kernel extension package. The **`library.c`** file automatically includes the **`library.h`** header file if it is not already included. If you have a project with multiple source code files, you should only include **library.c** once, but use the directive

```
#include "library.h"
```

in every additional source file that references the EtherSmart/WiFi functions.

If you have created resources such as web pages or images using the image converter file, then also use the directive

```
#include "image_data.h"   // if web resources have been defined
```

in each source code file. The **image_data.h** file contains macros that define the base xaddress and size of each string or image resource; these macro constants are typically passed as input parameters in the handler functions that serve out web pages.

To compile the C program, use the "make" icon in your C IDE (Integrated Development Environment) to create the *.dlf (download file) to be sent to the controller.

The **C_Ether_WiFi_demo_includer.txt** file in the **Ether_WiFi_WC_Code\Ether_Demo** directory loads the **install.txt** and **image_data.txt** files and the **.dlf** C compiler output file for the Ethernet/WiFi demo program. After the first load of this includer file, only the **.dlf** file needs to be sent to the controller after each run of the C compiler during program development. To send a file to the controller, use the "Send File" menu item in the QED Terminal.

Type **main** in the terminal window to execute the program; see the more detailed comments in the main portion of this document.

Note that all of the functions in the kernel extension are of the **_forth** type. While they are fully callable from C, there are two important restrictions. First, **_forth** functions may not be called as part of a parameter list of another**_forth** function.

Second, for all V4.xx operating system kernels that use the Fabius C compiler, **_forth** functions may not be called from within an interrupt service routine unless the instructions found in the file named **forthirq.c** in the compiler distribution are followed. The new PDQ line of controllers that use V6.xx kernels with the GCC compiler tool chain are not subject to this restriction.

Using the Driver Code with Forth

C programmers can skip this section.

After loading the **install.txt** and **image_data.txt** files and typing **COLD**, use the terminal to send the "**library.4th**" file to the Mosaic Controller (or **#include library.h** from the top of your Forth source code file). The **library.4th** file sets up a reasonable memory map and then defines the constants, structures, and name headers used by the EtherSmart/WiFi Wildcard kernel extension. This file leaves the memory map in the download map.

After **library.4th** has been loaded, the board is ready to receive your high level source code files. Be sure that your software doesn't initialize the memory management variables **DP**, **VP**, or **NP**, as this could cause memory conflicts. If you wish to change the memory map, edit the memory map commands at the top of the **library.4th** file itself. The definitions in **library.4th** share memory with your Forth code, and are therefore vulnerable to corruption due to a crash while testing. If you have problems after reloading your code, try typing **COLD** and reload everything starting with **library.4th**. It is very unlikely that the kernel extension runtime code itself (**install.txt**) or the resource file (**image_data.txt**) can become corrupted since these are stored in flash that is not typically accessed by code downloads.

The **Ether_WiFi_demo_includer.4th** file in the **Ether_WiFi_WC_Code\Ether_WiFi_Demo** directory loads the **install.txt** and **image_data.txt** files and the **Ether_WiFi_demo.4th** source file for the Ethernet/WiFi demo program. After the first load of this includer file, only the **Ether_WiFi_demo.4th** file needs to be sent to the controller after each source code editing cycle during program development. To send a file to the controller, use the "Send File" menu item in the Mosaic Terminal.

We recommend that your source code file end with the following sequence that is compatible with all Mosaic controller platforms shipped to date:

```
FIND WHICH.MAP      \ v4.xx: we're in download map:
IFTRUE          \ nesting is ok if endiftrues are next to each other
 XDROP
   4 PAGE.TO.FLASH
  5 PAGE.TO.FLASH
  6 PAGE.TO.FLASH
   STANDARD.MAP
  SAVE
OTHERWISE         \ for V6.xx kernels, store to shadow flash and save pointers
    SAVE.ALL .    \ for V6.xx. this takes some time, should print FFFF for success
ENDIFTRUE
```

This copies all loaded code from RAM to flash. The **SAVE** (V4.xx) or **SAVE.ALL** (V6.xx) command means that you can often recover from a crash and continue working by typing **RESTORE** (V4.xx) or **RESTORE.ALL** (V6.xx).

Appendix B: C Demo Program

Listing 1-19 Ether_WiFi_Demo.c program listing.

```
// this demonstration code is provided in source form.
// Top level functions:
// Ether_Web_Demo( ) // no input parameters; runs web pages, enables email, tunneling;
// WiFi_Web_Demo( ) // no input parameters; runs web pages, enables email, tunneling;
// Ether_Monitor_Demo( int modulenum ) // runs ethernet interactive monitor
// at the bottom of this file:
//        you can edit main to call the version of the web demo for your Wildcard
// these can be called interactively (from QEDTerminal)
// after Ether_Web_Demo or WiFi_Web_Demo is invoked:
// Tunnel_Test( ) // no input parameters, echoes data from Putty connection
// Email_Test( )  // no input parameters, returns error code.  Sends an email;
// Note: YOU MUST EDIT THIS FUNCTION's HARD-CODED IP ADDRESS and
// the hostname_str, sender_str, and recipient_str BEFORE CALLING!
// Make sure to edit the E_MODULENUM constant to match your hardware jumper settings.

// ************** #includes **************
#include <\mosaic\allqed.h>     // include all of the qed and C utilities
// choose the correct kernel version:
#include "Library\V4_4\library.h"       // for QScreen, QCard, Handheld controllers
#include "Library\V4_4\library.c"
// #include "Library\V4_09\library.h"   // for QVGA controller
// #include "Library\V4_09\library.c"
// include the header file that defines the base xaddress
// and 32-bit size for each resource:
#include "Resources\V4_4\image_headers.h" // for QScreen, QCard, Handheld controllers
// #include "Resources\V4_09\image_headers.h"   // for QVGA controller
// image_headers.h defines the following image and html string parameters:
// FORM_PAGE_HTML_XADDR
// FORM_PAGE_HTML_SIZE
// HOME_BODY_HTML_XADDR
// HOME_BODY_HTML_SIZE
// MOSAIC_LOGO_PNG_XADDR
// MOSAIC_LOGO_PNG_SIZE
// OPERA_CONFIG_HTML_XADDR
// OPERA_CONFIG_HTML_SIZE
#include "strdefs.h"     // defines STRING_XADDR(str_addr) macro; see comments below

// ************** USEFUL MACRO FOR STRINGS IN V4.xx C COMPILER **************
// the TO_XADDR defined in /mosaic/include/types.h
// transforms a separate 16-bit addr, page into a 32-bit xaddress:
// #define TO_XADDR(address,page)     ((xaddr) (((page)<<16)+ (0xFFFF & (address))))
// We want to substitute THIS_PAGE (also defined in types.h) for the page,
// as the V4.xx C compiler replicates the strings on each page;
// in most cases, the calling function's page is the same as the string page:
// #define STRING_XADDR(str_addr)
//               ((xaddr) TO_XADDR(((xaddr) str_addr), ((xaddr) THIS_PAGE)))

// ************** USEFUL CONSTANTS **************
// #define TASK_SIZE 0x400 // 0x400 = decimal 1024 = 1Kbyte = task size
#define CR_ASCII  0x0D    // ascii carriage return
#define CRLF_ASCII 0x0D0A // this is the standard eol sequence for tcp/ip services.
#define SMTP_PORT 25     // email Simple Mail Transfer Protocol destination port

// ***** IMPORTANT: SET MODULENUM TO MATCH HARDWARE JUMPERS *****************
#define E_MODULENUM      4 // *** SET THIS TO MATCH HARDWARE JUMPERS J1 AND J2! ***
// This specifies the modulenum for the high level functions in this file
// except for Ether_Monitor_Demo (see its comments).
// If EtherSmart is installed on module bus 0, E_MODULENUM = 0, 1, 2, or 3
// If EtherSmart is installed on module bus 1, E_MODULENUM = 4, 5, 6, or 7
// That is, the bus specifies the top bit of a 3-bit modulenum,
// and [J2, J1] specifies the remaining 2 bits of the modulenum.
// Example: On module bus 0, if neither jumper cap is installed: E_MODULENUM = 0
// Example: On module bus 0, if both jumper caps are installed: E_MODULENUM = 3
// Example: On module bus 1, if J2 is installed but J1 is not: E_MODULENUM = 6

// ************** SETUP ETHERNET TASK TO RUN WEB AND TUNNELING SERVICES **************

TASK ether_control_task;  // 1 Kbyte per task area

_Q void Ether_Task_Setup_Default( void )
// performs full initialization of the Ether_Info struct and mailboxes for the
// modulenum specified by the E_MODULENUM constant, and
// builds and activates an ethernet control task to service the EtherSmart Wildcard
{
  // NEXT_TASK = TASKBASE; // empties task loop; comment out if other tasks allowed
  Ether_Task_Setup( &ether_control_task, E_MODULENUM);
}

_Q void WiFi_Task_Setup_Default( void )
// same as Ether_Task_Setup_Default, but marks Wildcard as WiFi
// performs full initialization of the Ether_Info struct and mailboxes for the
// modulenum specified by the E_MODULENUM constant, and
// builds and activates an ethernet control task to service the WiFi Wildcard
{
  // NEXT_TASK = TASKBASE; // empties task loop; comment out if other tasks allowed
  WiFi_Task_Setup( &ether_control_task, E_MODULENUM);
}

// ***************** INTERACTIVE CONFIGURATION AND REPORTING ************

_Q int Ether_Set_Defaults( void )
// works for xport or wiport (ethernet/wifi)
// call this AFTER calling main() or Ether_Web_Demo or Ether_Task_Setup_Default()
// or WiFi_Web_Demo or WiFi_Task_Setup_Default.
// sets mosaic factory defaults; returns error code
// sets local IP and gateway to 0.0.0.0 = unassigned, so IP address
// gets assigned via DHCP (Dynamic Host Configuration Protocol) by the LAN's gateway.
// see user guide for more information.
{ printf("\rSetting defaults...\r");
  Ether_XPort_Defaults(E_MODULENUM); // works for xport or wiport (ethernet/wifi)
  return((int) Ether_Await_Response(E_MODULENUM)); // error code is in lsword
}

_Q int Ether_Set_Local_IP(int my_ip1, int my_ip2, int my_ip3,  int my_ip4)
// call this AFTER calling main() or Ether_Web_Demo or Ether_Task_Setup_Default()
// or WiFi_Web_Demo or WiFi_Task_Setup_Default.
// sets the IP address of the EtherSmart Wildcard specified by E_MODULENUM as:
//   ip1.ip2.ip3.ip4
// For example, to set IP address to 10.0.1.22, pass to this function the parameters:
//   10 0 1 22
// returns error code
// NOTES: type DECIMAL at the monitor before invoking this function interactively!
//        The input types are declared as int to simplify interactive calling,
//        as the interactive debugger would require char specifiers before each input
//        parameter if the char type were used.
// NOTE: assigning a nonzero IP address disables DHCP!
{ printf("\rSetting local IP address...\r");
  Ether_Local_IP(my_ip1, my_ip2, my_ip3,  my_ip4, E_MODULENUM);
  Ether_XPort_Update(E_MODULENUM);  // works for xport or wiport (ethernet/wifi)
  return((int) Ether_Await_Response(E_MODULENUM)); // error code is in lsword
}

_Q void Ether_IP_Report(void)
// call this AFTER calling main() or Ether_Web_Demo or Ether_Task_Setup_Default()
// or WiFi_Web_Demo or WiFi_Task_Setup_Default.
// takes 7 seconds to execute, so be patient.
// Report is of the form:
// IP 010.000.001.019 GW 010.000.001.022 Mask 255.255.255.000
// which summarizes the IP address, gateway address, and netmask, respectively.
{ Ether_IP_Info_Report(E_MODULENUM);
}
```

```
_Q void Ether_Ping(int remote_ip1, int remote_ip2, int remote_ip3,  int remote_ip4)
// call this AFTER calling main() or Ether_Web_Demo or Ether_Task_Setup_Default()
// or WiFi_Web_Demo or WiFi_Task_Setup_Default.
// on error, prints " Couldn't enter monitor mode!" or " No response from remote".
// takes thirteen seconds to execute, so be patient.
// Report is of the form (summarizes response time from specified remote host):
// Seq 001 time 10ms
// Seq 002 time 10ms
// Seq 003 time 10ms
// Seq 004 time 10ms
// Seq 005 time 10ms
// Seq 006 time 10ms
// NOTES: type DECIMAL at the monitor before invoking this function interactively!
//        The input types are declared as int to simplify interactive calling,
//        as the interactive debugger would require char specifiers before each input
//        parameter if the char type were used.
{  Ether_Ping_Report(remote_ip1, remote_ip2, remote_ip3, remote_ip4, E_MODULENUM);
}

// ***************** REVECTORED SERIAL VIA TCP/IP ************
// simplest approach:
// simply store the correct modulenum into the global var named ether_revector_module,
// then call Ether_Serial_Revector
// This will run the default (startup) task through the specified ethersmart wildcard.
// Use Putty "rlogin" to connect to the local port (typically = 80) at the
// specified local IP address, and you're talking to the QED monitor via TCP/IP.
// To revert to standard serial operation via QED Term,
//  type COLD to revert to standard serial,
// then from QEDTerm type RESTORE to bring back access to all compiled routines,
// then continue communications using QEDTerm.
// If you want to maintain serial communications via QEDTerm with the default task
// while running a separate task with I/O revectored via the Ethersmart Wildcard,
// then build and activate a task using Ether_Monitor as the activation routine.
// Use Putty "rlogin" to connect to the local port (typically = 80)
// at the specified local IP address, and you're talking to the task via ethernet.

TASK ether_montask;  // 1 Kbyte per task area

_Q void Ether_Monitor_Demo( int modulenum )
// works for xport or wiport (ethernet/wifi)
// builds and activates a monitor task for the specified module.
// this function expects the modulenum as an input parameter so that
// you can run 2 installed Ethersmart wildcards at once:
// one wildcard can run the standard services (web, email, tunneling)
// while the other one (running Ether_Monitor_Demo) is used to download code
// and control program development and debugging.
// To do this, first call Ether_Web_Demo or WiFi_Web_Demo
// (which runs services on the E_MODULENUM wildcard),
// then call this Ether_Monitor_Demo function with a different modulenum.
{  ether_revector_module = modulenum;    // set variable to specify revectored module
   SERIAL_ACCESS = RELEASE_ALWAYS; // allow sharing of serial ports
   // NEXT_TASK = TASKBASE; // empties task loop; comment out if other tasks allowed
   BUILD_C_TASK(0, 0, &ether_montask );
   ACTIVATE(Ether_Monitor, &ether_montask );
}

// ******************* SERIAL TUNNELING TEST **************

_Q void Tunnel_Test( void )
// works for xport or wiport (ethernet/wifi)
// call after doing Ether_Task_Setup_Default or WiFi_Task_Setup_Default
// Waits for incoming non-web connection on this module's IP address on port 80
// (the default local port), and examines the linefeed-delimited first line to see
// if it starts with GET; if not, it accepts it as a passive non-web connection.
// So: don't type GET as the first characters of the connection
//  (you'll confuse the system).
// The easiest way to open a connection is via Putty, using the "raw" mode.
// Note: If you use "telnet" mode, you'll see some
// garbage characters in the first line; these are
// control bytes with the msbit set that are accepted by the connection manager.
{  int num_received=0, inbufcount=0, i=0, numlines=0;
   char c;
   int timeout = 20000; // 20 second timeout for testing; sets delay til function exit
   xaddr ether_inbuf_base = Ether_Inbuf(E_MODULENUM);
   uint ether_inbuf_size =  Ether_Inbufsize(E_MODULENUM);
   xaddr http_inbuf_base = HTTP_Inbuf(E_MODULENUM);
   char* prompt = "Line was received>";
   int prompt_length = strlen(prompt);
   printf("\rWaiting for connection (type a carriage return from QEDTerm to
   abort)...\r");
   printf("Suggestion: To connect, use Putty in 'raw' mode, specify IP address, port
   80,\r");
   printf("and type a carriage return once the session window opens.\r");
   do
   { PauseOnKey;          // abort on CR, pause/resume on other keys
     Ether_Check_Response(E_MODULENUM);  // keep mailbox clean, ignore messages
   }
   while(!Ether_Passive_Non_Web_Connection(E_MODULENUM));// wait for tunnel connection
   printf("\rAn incoming connection has been accepted;\r");
   printf("each incoming line will be printed to the serial terminal, and a\r");
   printf("response prompt will be sent to the ethernet terminal.\r");
   printf("To exit this routine, wait 20 seconds without typing a line.\r");
// the first line is in HTTP_Inbuf (because webserver had to check its identity)...
// count is in 1st 2bytes of buffer
   inbufcount = (FetchInt(http_inbuf_base)); // count stored in first 2bytes of buffer
   for( i=0; i<inbufcount; i++)
   { c = FetchChar(http_inbuf_base + 2 + i); // skip 2byte count,get char
     Emit(c);
   }       // type first line including terminating crlf to local serial terminal
   do              // loop and echo additional lines til timeout...
   {
   Ether_Get_Line(ether_inbuf_base,ether_inbuf_size,CR_ASCII,1,1,timeout,E_MODULENUM);
// xlbuf,maxchars,eol,discard.alt.eol?,no.msbitset?,timeout_msec,module
// get 1 line into counted ether_inbuf (count is stored in first 2 bytes)
    num_received = (int) Ether_Await_Response(E_MODULENUM);    // get lsword of result
    inbufcount = (FetchInt(ether_inbuf_base));
    for( i=0; i<inbufcount; i++)
    { c = FetchChar(ether_inbuf_base + 2 + i); // skip 2byte count,get char
      Emit(c);
    }      // type line including terminating crlf to local serial terminal
    if(num_received) // send prompt and wait for response if chars were received
    { Ether_Send_Buffer( STRING_XADDR(prompt), prompt_length, timeout, E_MODULENUM);
      Ether_Await_Response(E_MODULENUM); // wait for result, ignore rtn value
      numlines++;                                               // bump numlines
    }
   }
   while(num_received);          // until no chars rcvd due to timeout or connection
   close
   Ether_Disconnect_Flush(E_MODULENUM);  // clean up
   Ether_Await_Response(E_MODULENUM);    // synchronize to result mailbox before exit
   printf("\r\nConnection terminated.\r\n");
}

// ********************* EMAIL  TEST ***************************

// NOTE: Edit these to specify valid host, sender, and recipient:
char* hostname_str = "wildcard.yourdomain.com";        // hostname string
char* sender_str =  "niceguy@yourdomain.com";   // email sender string
char* recipient_str = "notso_niceguy@yourdomain.com";  // email recipient string
char* email_body_str =   // email_body string
"Subject: EtherSmart/WiFi Email Test\r\n\
This is a test email from the EtherSmart/WiFi Wildcard.\r\n\
Emails can of course have multiple lines...";

_Q int Email_Test( void )
```

```
// works for xport or wiport (ethernet/wifi)
// call after doing Ether_Task_Setup_Default or WiFi_Task_Setup_Default
// smtp email test routine.
// Note: YOU MUST EDIT THIS FUNCTION'S HARD-CODED IP ADDRESS
// AND the hostname_str, sender_str, and recipient_str BEFORE CALLING!
// Otherwise, it will not work on your system.
// error= 0 on success,
// or error_no_response = 0x10, or the smtp 3-digit decimal error code
{ xaddr scratchbuf = Ether_Inbuf( E_MODULENUM );              // scratchpad xbuffer
  Ether_Send_Email(
  STRING_XADDR(email_body_str), strlen(email_body_str), // subject plus 2-line email
  STRING_XADDR(hostname_str), strlen(hostname_str),
  STRING_XADDR(sender_str), strlen(sender_str),
  STRING_XADDR(recipient_str), strlen(recipient_str),
  scratchbuf,           // scratchpad xbuffer
  10, 10, 10, 1,                // IMPORTANT: ENTER YOUR GATEWAY'S IP ADDRESS HERE!
  SMTP_PORT,                    // port 25 = standard mailserver destination port
  10000,                        // use a 10 second timeout (10,000 msec) for testing
  E_MODULENUM);         // specify module number (must match hardware jumpers)
  return((int) Ether_Await_Response(E_MODULENUM)); // error code is in lsword
}

// *********************** WEBSERVER DEMO (Ethersmart Web Pages) ***************

// time announcement string for home page:
char* time_announce_str = "seconds since system initialization.</body></html>";
// form_response page strings:
char* form_response_str = "<html><head><title>Form Response</title></head><body>\r\n\
<H3>Thanks for your response!</H3><p><p>It must be great to be a ";
char* person_str = "person";
char* name_announce_str = " with a name like ";
char* yours_str = "yours";
char* good_to_hear_str = ". It's good to hear that ";
char* rainbow_str = "rainbow";
char* favorite_str = " is your favorite color!</body></html>";
// url strings:
char* slash_url_str = "/"; // synonym for home page url
char* index_url_str = "/index.html"; // home page url
char* opera_url_str = "/opera_configuration.html"; // opera page url
char* form_entry_url_str = "/form_entry.html"; // form entry page url
char* form_response_url_str = "/form_response.cgi"; // form response page url
char* logo_response_url_str = "/logo_response.png"; // logo response page url
// note: in form_page_str, we mark one of the radio buttons as checked; otherwise,
// the browser can skip the field entirely,
// then we would have to match the fieldnames!
char seconds_str[16];    // will hold elapsed seconds string

void Home_Page( int modulenum )
// this is the handler function for the /index.html and the / URLs.
// sends the http header with dynamic (because of changing time stamp) text/html
// content type,  followed by home_body_str text and the elapsed seconds since the
// system was initialized.
{ ulong elapsed_sec;
  xaddr http_outbuf_base = HTTP_Outbuf(modulenum);
  uint http_outbuf_size =  HTTP_Outbufsize(modulenum);
  HTTP_Put_Header(http_outbuf_base, http_outbuf_size); // http header->outbuf
  HTTP_Put_Content_Type(http_outbuf_base,http_outbuf_size,1,HTTP_TEXT_HTML_CONTENT);
// params: xlbuf,maxbufsize,dynamic?,content_type
// dynamic text/html content ->outbuf;header is done
  HTTP_Send_LBuffer(http_outbuf_base, modulenum); // send header,ignore #bytes_sent
  HTTP_Send_Buffer(HOME_BODY_HTML_XADDR, HOME_BODY_HTML_SIZE, modulenum);
// send home_body.html, ignore #bytes_sent
  StoreInt(0, http_outbuf_base); // reset lcount to start fresh in buffer
  elapsed_sec = ReadElapsedSeconds();  // get elapsed time, used by sprintf...
  HTTP_Outbuf_Cat((xaddr) seconds_str,(int)
  sprintf(seconds_str,"%ld",elapsed_sec),modulenum);
// params: xstring_to_add,cnt_to_add,module
// convert seconds to string, add line & crlf to longstring, update lcnt
```

```
 HTTP_Outbuf_Cat(STRING_XADDR(time_announce_str),strlen(time_announce_str),modulenum);
// add to lbuffer: " seconds since system initialization.</body></html>"
  HTTP_Send_LBuffer(http_outbuf_base,modulenum); // send elapsed time
}

void Opera_Config_Page( int modulenum )
// this is the handler function for the /opera_configuration.html URL.
// sends the http header with static text/html content type,
// followed by opera_config.html text
{ xaddr http_outbuf_base = HTTP_Outbuf(modulenum);
  uint http_outbuf_size =  HTTP_Outbufsize(modulenum);
  HTTP_Put_Header(http_outbuf_base, http_outbuf_size); // http header->outbuf
  HTTP_Put_Content_Type(http_outbuf_base,http_outbuf_size,0,HTTP_TEXT_HTML_CONTENT);
// params: xlbuf,maxbufsize,dynamic?,content_type
// static text/html content ->outbuf;header is done
  HTTP_Send_LBuffer(http_outbuf_base, modulenum); // send header,ignore #bytes_sent
  HTTP_Send_Buffer(OPERA_CONFIG_HTML_XADDR, OPERA_CONFIG_HTML_SIZE, modulenum);
}                                      // send opera_config.html, ignore #bytes_sent

void Form_Entry_Page( int modulenum )
// responds to url= /form_entry.html
// places FORM_TEXT into the buffer as a longstring after the http header.
// we send directly from compiled string in flash to avoid copy into ether_out buffer;
// this method is ideal for serving static text.
{ xaddr http_outbuf_base = HTTP_Outbuf(modulenum);
  uint http_outbuf_size =  HTTP_Outbufsize(modulenum);
  HTTP_Put_Header(http_outbuf_base, http_outbuf_size); // http header->outbuf
  HTTP_Put_Content_Type(http_outbuf_base,http_outbuf_size,0,HTTP_TEXT_HTML_CONTENT);
// params: xlbuf,maxbufsize,dynamic?,content_type
// static text/html content ->outbuf;header is done
  HTTP_Send_LBuffer(http_outbuf_base, modulenum); // send header, ignore #bytes_sent
  HTTP_Send_Buffer(FORM_PAGE_HTML_XADDR, FORM_PAGE_HTML_SIZE, modulenum);
}                                      // send form_page.html, ignore #bytes_sent

void Form_Response_Page  ( int modulenum )
// this is the handler function that responds to url= /form_response.cgi
// incoming request from browser looks like this:
// GET /form_response.cgi?classification=<man/woman/child>&name_id=<namestring>
// &color=<red/blue/yellow/green>
// We respond:
// It must be great to be a <man/woman/child/person>
//  with a name like <namestring/yours>.
// It's good to hear that <red/blue/yellow/green/rainbow> is your favorite color.
{ xaddr http_outbuf_base = HTTP_Outbuf(modulenum);
  uint http_outbuf_size =  HTTP_Outbufsize(modulenum);
  xaddr http_value_ptr = HTTP_Value_Ptr(modulenum); // load value,cnt for 1st field
  uint http_value_count =  HTTP_Value_Count(modulenum);
  xaddr next_http_value_ptr; // used for look-ahead field to avoid unescape problems
  uint next_http_value_count; // used for look-ahead field to avoid unescape problems
  HTTP_Put_Header(http_outbuf_base, http_outbuf_size); // http header->outbuf
  HTTP_Put_Content_Type(http_outbuf_base,http_outbuf_size,0,HTTP_TEXT_HTML_CONTENT);
// params: xlbuf,maxbufsize,dynamic?,content_type
// static text/html content ->outbuf;header is done
  HTTP_Send_LBuffer(http_outbuf_base, modulenum); // send header, ignore #bytes_sent
  HTTP_Send_Buffer(STRING_XADDR(form_response_str),strlen(form_response_str),
  modulenum);
// send form_response_str, ignore #bytes_sent
  StoreInt(0, http_outbuf_base); // reset lcount to start fresh in buffer
  if(http_value_count)   // if first field was entered, add it to response
  HTTP_Outbuf_Cat(http_value_ptr, http_value_count, modulenum);
  else                                 // else add "person" to response
  HTTP_Outbuf_Cat(STRING_XADDR(person_str), strlen(person_str), modulenum);
  HTTP_Outbuf_Cat(STRING_XADDR(name_announce_str), strlen(name_announce_str),
  modulenum);
// add " with a name like " to response
  HTTP_To_Next_Field(modulenum); // go to name_id field, ignore #chars_advanced
  http_value_ptr = HTTP_Value_Ptr(modulenum);  // load value,cnt for name_id field
  http_value_count =  HTTP_Value_Count(modulenum);
```

```
   HTTP_To_Next_Field(modulenum); // go to color field, ignore #chars_advanced
   next_http_value_ptr = HTTP_Value_Ptr(modulenum);// load value,cnt for color field
   next_http_value_count =  HTTP_Value_Count(modulenum);
   if(http_value_count)   // if a name was entered, process it and add it to response
   { HTTP_Plus_To_Space(http_value_ptr, http_value_count); // rid of +'s in name text
    http_value_count = HTTP_Unescape(http_value_ptr, http_value_count);
    HTTP_Outbuf_Cat(http_value_ptr, http_value_count, modulenum);
   }
   else                                  // else add "yours" to response
   HTTP_Outbuf_Cat(STRING_XADDR(yours_str), strlen(yours_str), modulenum);
   HTTP_Outbuf_Cat(STRING_XADDR(good_to_hear_str), strlen(good_to_hear_str),
   modulenum);
// add ". It's good to hear that " to response
   if(next_http_value_count)     // if color field was entered, add it to response
   HTTP_Outbuf_Cat(next_http_value_ptr, next_http_value_count, modulenum);
   else                                  // else add "rainbow" to response
   HTTP_Outbuf_Cat(STRING_XADDR(rainbow_str), strlen(rainbow_str), modulenum);
   HTTP_Outbuf_Cat(STRING_XADDR(favorite_str), strlen(favorite_str), modulenum);
// add " is your favorite color!</body></html>" to response
   HTTP_Send_LBuffer( http_outbuf_base, modulenum); // send response
}

void Logo_Response_Page  ( int modulenum )
// this is the handler function that responds to url= /logo_response.png
// assumes that logo.s2 has been loaded into memory. serves it as static image.
{  xaddr http_outbuf_base = HTTP_Outbuf(modulenum);
   uint http_outbuf_size =  HTTP_Outbufsize(modulenum);
   int http_header_size;
   HTTP_Put_Header(http_outbuf_base, http_outbuf_size); // http header->outbuf
   HTTP_Put_Content_Type(http_outbuf_base,http_outbuf_size,0,HTTP_IMAGE_PNG_CONTENT);
// params: xlbuf,maxbufsize,dynamic?,content_type
// static png image type->outbuf, cnt in first 2bytes; header's done
   http_header_size = FetchInt(http_outbuf_base);// get size of header we created
   HTTP_Send_2Buffers(http_outbuf_base+2, http_header_size,
   MOSAIC_LOGO_PNG_XADDR, MOSAIC_LOGO_PNG_SIZE, modulenum);
// params: xbuf1,cnt1,xbuf2,count2,modulenum
}                                        // send header&image, ignore numbytes_sent

#ifdef      __FABIUS__
#pragma option init=.doubleword    // declare 32-bit function pointers in code area
#include </mosaic/gui_tk/to_large.h>
#endif    // __FABIUS__
xaddr (*home_page_ptr)(void)          = Home_Page;
xaddr (*opera_config_page_ptr)(void)  = Opera_Config_Page;
xaddr (*form_entry_page_ptr)(void)    = Form_Entry_Page;
xaddr (*form_response_page_ptr)(void) = Form_Response_Page;
xaddr (*logo_response_page_ptr)(void) = Logo_Response_Page;
#ifdef      __FABIUS__
#include </mosaic/gui_tk/fr_large.h>
#pragma option init=.init       // return the initialized variable area to RAM;
#endif    // __FABIUS__

int Web_Handler_Installation( int modulenum )
// call this after Ether_Task_Setup_Default
// point browser to raw ip or to ip/index.html to see the home web page.
// urls are case sensitive. any other url's serve out: page not found.
// returns nonzero error if too many handlers were added
// (limited by AUTOSERVE_DEFAULT_ROWS passed to ether_init)
{  int error = 0; // we'll OR error results together and return final result
   error |= HTTP_Add_Handler(STRING_XADDR(slash_url_str), strlen(slash_url_str),
          home_page_ptr, modulenum);
   error |= HTTP_Add_Handler(STRING_XADDR(index_url_str), strlen(index_url_str),
          home_page_ptr, modulenum);
   error |= HTTP_Add_Handler(STRING_XADDR(opera_url_str), strlen(opera_url_str),
          opera_config_page_ptr, modulenum);
   error |= HTTP_Add_Handler(STRING_XADDR(form_entry_url_str),
   strlen(form_entry_url_str), form_entry_page_ptr, modulenum);
   error |= HTTP_Add_Handler(STRING_XADDR(form_response_url_str),
          strlen(form_response_url_str), form_response_page_ptr, modulenum);
  error |= HTTP_Add_Handler(STRING_XADDR(logo_response_url_str),
          strlen(logo_response_url_str), logo_response_page_ptr, modulenum);
  InitElapsedTime();// start at zero so home page reports correct elapsed time
  return(error);
}

_Q void Ether_Web_Demo( void )
// for EtherSmart Wildcard: point browser to raw ip or to
//  ip/index.html to see the home web page.  urls are case sensitive.
// any other url's serve out: page not found.
// returns nonzero error if too many handlers were added
// (limited by AUTOSERVE_DEFAULT_ROWS passed to ether_init)
{  Ether_Task_Setup_Default();
   Web_Handler_Installation(E_MODULENUM);// ignore too_many_handlers? flag
}

_Q void WiFi_Web_Demo( void )
// for WiFi Wildcard: point browser to raw ip or to
//  ip/index.html to see the home web page.  urls are case sensitive.
// any other url's serve out: page not found.
// returns nonzero error if too many handlers were added
// (limited by AUTOSERVE_DEFAULT_ROWS passed to ether_init)
{  WiFi_Task_Setup_Default();
   Web_Handler_Installation(E_MODULENUM);// ignore too_many_handlers? flag
}

void main( void )
{                 // comment in the correct version for your wildcard:
Ether_Web_Demo();
// WiFi_Web_Demo();
}
```

Appendix C: C Remote Front Panel Demo Program

Listing 1-20 Ether_GUI_Demo.c program listing.

```
// this demonstration code is provided in source form.
// Top level function:
// Ether_GUI_Web_Demo( ) \ no input parameters; runs gui and web page on EtherSmart WC
// WiFi_GUI_Web_Demo( ) \ no input parameters; runs gui and web page on WiFi WC
// at the bottom of this file:
//  you can edit main to call the version of the web GUI demo for your Wildcard

// *************** IMPORTANT: SET MODULENUM TO MATCH HARDWARE JUMPERS ******************

// recall: modulenum 0 is reserved on the QScreen!

#define E_MODULENUM 4   // ******** SET THIS TO MATCH HARDWARE JUMPERS J1 AND J2! *****

// This specifies the modulenum for the high level functions in this file
// except for Ether_Monitor_Demo (see its comments).

// If EtherSmart is installed on module bus 0, E_MODULENUM = 0, 1, 2, or 3
// If EtherSmart is installed on module bus 1, E_MODULENUM = 4, 5, 6, or 7
// That is, the bus specifies the top bit of a 3-bit modulenum,
// and [J2, J1] specifies the remaining 2 bits of the modulenum.
// Example: On module bus 0, if neither jumper cap is installed: E_MODULENUM = 0
// Example: On module bus 0, if both jumper caps are installed: E_MODULENUM = 3
// Example: On module bus 1, if J2 is installed but J1 is not: E_MODULENUM = 6

// ************** #includes **************

#include <\mosaic\allqed.h>     // include all of the qed and C utilities

// choose the correct controller product, comment in/out the appropriate #includes:

#include "Library\esmart_gui_qvga\library.h" // for QVGA controller
#include "Library\esmart_gui_qvga\library.c"

// #include "Library\esmart_gui_qscreen\library.h" // for QScreen controller
// #include "Library\esmart_gui_qscreen\library.c"



// include the header file that defines the base xaddress and 32-bit size for each resource:
#include "Resources\qvga\image_headers.h" // for QVGA controller
// #include "Resources\qscreen\image_headers.h" // for QScreen controller


#include "guidemo_qvga.c"     // gui demo program for qvga controller
// #include "guidemo_qscreen.c"  // gui demo program for qscreen controller



#include "strdefs.h" // defines STRING_XADDR(str_addr) macro; see comments below

// ************** USEFUL MACRO FOR STRINGS IN V4.xx C COMPILER **************

// the TO_XADDR defined in /mosaic/include/types.h
// transforms a separate 16-bit addr, page into a 32-bit xaddress:
// #define TO_XADDR(address,page)      ((xaddr) (((page)<<16)+ (0xFFFF & (address))))

// We want to substitute THIS_PAGE (also defined in types.h) for the page,
// as the V4.xx C compiler replicates the strings on each page;
// therefore, in most cases, the calling function's page is the same as the string page:
```

```
// #define STRING_XADDR(str_addr) ((xaddr) TO_XADDR(((xaddr) str_addr), ((xaddr)
THIS_PAGE)))



// ************** USEFUL CONSTANTS **************

#ifdef      __FABIUS__
#define BITMAP_BUFFER 0x020000   // if V4.xx, put on pg 2
#else
#define BITMAP_BUFFER 0x148000   // if V6.xx, put on pg 0x14
#endif   // __FABIUS__

#define BITMAP_MAXBYTES 0x8000 // 32K max, plenty for monochrome displays
#define BITMAP_SCREEN_FORMAT 1 // used by Screen_To_Image, Simulated_Touch_To_Image




// ************** SETUP ETHERNET TASK TO RUN WEB AND TUNNELING SERVICES **************

TASK ether_control_task;  // 1 Kbyte per task area

_Q void Ether_Task_Setup_Default( void )
// performs full initialization of the ether_info struct and mailboxes for the
// specified EtherSmart modulenum, and
// builds and activates an ethernet control task to service the xport
{
 // NEXT_TASK = TASKBASE;    // empties task loop; comment out if other tasks are allowed
   Ether_Task_Setup( &ether_control_task, E_MODULENUM);
}

_Q void WiFi_Task_Setup_Default( void )
// performs full initialization of the ether_info struct and mailboxes for the
// specified WiFi modulenum, and
// builds and activates an ethernet control task to service the xport
{
 // NEXT_TASK = TASKBASE;    // empties task loop; comment out if other tasks are allowed
   WiFi_Task_Setup( &ether_control_task, E_MODULENUM);
}

// ********************* URL STRINGS *************************

char* slash_url_str = "/"; // synonym for home page url

char* index_url_str = "/index.html"; // home page url

char* screen_image_str = "/screen_image.bmp"; // screen image url

char* gui_response_str = "/gui_response.cgi"; // gui response url



// *********************** WEBSERVER ***************************

// the following is created using the Image Converter program which creates 2 constants:
// GUI_RESPONSE_TEXT_HTML_XADDR
// GUI_RESPONSE_TEXT_HTML_SIZE

// <html><head><title>EtherSmart/GUI Remote Front Panel</title></head>
// <body>
// <H3>EtherSmart/GUI Remote Front Panel</H3><p>
// <H4>Click on the screen image to operate the touchscreen remotely</H3><p><p>
// <a href="/gui_response.cgi"><img src="/screen_image.bmp" ismap></a>
// </body></html>

// make sure to declare gui_response.cgi as Cache-Control: no-cache (dynamic content).
// Notes: this works very nicely with Opera, the recommended browser.
```

```
void Screen_Image_Response( int modulenum )
// handler for /screen_image.bmp url. serves the screen image.
// assumes that Simulated_Touch_To_Image has run.
// xbuffer contains 32-bit count (NOTE!) followed by image data.
// marks as no-cache (always reloads).
// NOTE: this function is NOT a gui handler; it should be posted by http_add_handler.
{  xaddr http_outbuf_base = HTTP_Outbuf(modulenum);
   uint http_outbuf_size =  HTTP_Outbufsize(modulenum);
   uint image_size = FetchInt((xaddr) (BITMAP_BUFFER+2));   // get size of image we created
   uint http_header_size;
   HTTP_Put_Header(http_outbuf_base, http_outbuf_size);  // http header->outbuf
   HTTP_Put_Content_Type(http_outbuf_base, http_outbuf_size, 1, HTTP_IMAGE_BITMAP_CONTENT);
                  // params: xlbuf,maxbufsize,dynamic?,content_type
                  // dynamic bitmap image content ->outbuf;header is done
   http_header_size = FetchInt(http_outbuf_base);  // get size of http header we created
   HTTP_Send_2Buffers(http_outbuf_base+2, http_header_size,
      (xaddr) (BITMAP_BUFFER+4), image_size, modulenum);
}                 // send http header and bitmap image

// from types.h, used in GUI_Response to hold HTTP_Imagemap x,y results.
// typedef union
// {  ulong int32;
//     struct
//     {   int    msInt;
//         int    lsInt;
//     } twoNums;
// }    TWO_INTS;

void GUI_Response( int modulenum )
// a clickable ismap version of the screen image that runs the gui toolkit simulated touch.
// url = /gui_response.cgi
// this routine is called via ether_check_gui which runs in the application task
// that runs Service_GUI_Events.  it is posted using http_gui_add_handler.
{  xaddr http_outbuf_base = HTTP_Outbuf(modulenum);
   uint http_outbuf_size =  HTTP_Outbufsize(modulenum);
   uint http_header_size, x, y;
   TWO_INTS x_and_y;
   x_and_y = HTTP_Imagemap( modulenum); // get mouseclick x,y coordinates
   x = x_and_y.twoNums.msInt; // get x from most significant 16bits
   y = x_and_y.twoNums.lsInt; // get y from least significant 16bits
   Simulated_Touch_To_Image(BITMAP_BUFFER, BITMAP_MAXBYTES, BITMAP_SCREEN_FORMAT, x, y);
   HTTP_Put_Header(http_outbuf_base, http_outbuf_size);  // http header->outbuf
   HTTP_Put_Content_Type(http_outbuf_base, http_outbuf_size, 0, HTTP_TEXT_HTML_CONTENT);
                  // params: xlbuf,maxbufsize,dynamic?,content_type
                  // dynamic bitmap image content ->outbuf;header is done
   http_header_size = FetchInt(http_outbuf_base);  // get size of http header we created
   HTTP_GUI_Send_2Buffers(http_outbuf_base+2, http_header_size,
      GUI_RESPONSE_TEXT_HTML_XADDR, GUI_RESPONSE_TEXT_HTML_SIZE, modulenum);
}                 // send http header and content

_Q int Init_Screen_Image( void )
// if return value=0, image has been created
{  return Screen_To_Image((xaddr) BITMAP_BUFFER, BITMAP_MAXBYTES, BITMAP_SCREEN_FORMAT);
}


void  Remote_Panel_Start( int modulenum )
// serves an initial clickable ismap version of the screen image.
// url = /index.html
// puts initial version of screen into bitmap_buffer.
// this routine is called via ether_check_gui which runs in the application task
// that runs Service_GUI_Events.  it is posted using http_gui_add_handler
{  xaddr http_outbuf_base = HTTP_Outbuf(modulenum);
   uint http_outbuf_size =  HTTP_Outbufsize(modulenum);
   int http_header_size;
   HTTP_Put_Header(http_outbuf_base, http_outbuf_size);  // http header->outbuf
   HTTP_Put_Content_Type(http_outbuf_base, http_outbuf_size,0, HTTP_TEXT_HTML_CONTENT);
                  // params: xlbuf,maxbufsize,dynamic?,content_type
                  // static html type->outbuf, cnt in first 2bytes; header's done
   Init_Screen_Image(); // ignore return value; image has been created
   http_header_size = FetchInt(http_outbuf_base);  // get size of http header we created
   HTTP_GUI_Send_2Buffers(http_outbuf_base+2, http_header_size,
            GUI_RESPONSE_TEXT_HTML_XADDR, GUI_RESPONSE_TEXT_HTML_SIZE, modulenum);
                  // params: xbuf1,cnt1,xbuf2,count2,modulenum
}                 // send header&content, ignore numbytes_sent

#ifdef      __FABIUS__
#pragma option init=.doubleword    // declare 32-bit function pointers in code area
#include </mosaic/gui_tk/to_large.h>
#endif   // __FABIUS__

xaddr (*screen_image_response_ptr)(void) = Screen_Image_Response;
xaddr (*gui_response_ptr)(void)          = GUI_Response;
xaddr (*remote_panel_start_ptr)(void)    = Remote_Panel_Start; // home page


#ifdef      __FABIUS__
#include </mosaic/gui_tk/fr_large.h>
#pragma option init=.init       // return the initialized variable area to RAM;
#endif   // __FABIUS__


int Install_GUI_Web_Handlers( int modulenum )
// call this after ETHER_Task_Setup_DEFAULT  point browser to raw ip or to
//  ip/index.html to see the home GUI web page = remote front panel.
// urls are case sensitive.
// any other url's serve out: page not found.
// returns nonzero error if too many handlers were added
// (limited by AUTOSERVE_DEFAULT_ROWS passed to ether_init)
// Some of the gui web handlers in this example are handled by the application task.
{  int error = 0; // we'll OR error results together and return final result
   error |= HTTP_Add_GUI_Handler(STRING_XADDR(slash_url_str), strlen(slash_url_str),
            remote_panel_start_ptr, modulenum);
   error |= HTTP_Add_GUI_Handler(STRING_XADDR(index_url_str), strlen(index_url_str),
            remote_panel_start_ptr, modulenum);
   error |= HTTP_Add_Handler(STRING_XADDR(screen_image_str), strlen(screen_image_str),
            screen_image_response_ptr, modulenum);
   error |= HTTP_Add_GUI_Handler(STRING_XADDR(gui_response_str), strlen(gui_response_str),
            gui_response_ptr, modulenum);
   InitElapsedTime();      // start at zero so home page reports correct elapsed time
   return(error);
}

_Q void Ether_GUI_Web_Demo( void )
{  Ether_Task_Setup_Default();// init EtherSmart WC, start ethernet task
   Install_GUI_Web_Handlers(E_MODULENUM);    // setup web; ignore error flag
   GUI_Demo();    // start the gui demo on local touchscreen
}                   // NOTE: for QScreen, GUI_Demo MUST call Globalize_TVars(TVARS);

_Q void WiFi_GUI_Web_Demo( void )
{  WiFi_Task_Setup_Default(); // init WiFi WC, start ethernet task
   Install_GUI_Web_Handlers(E_MODULENUM);    // setup web; ignore error flag
   GUI_Demo();    // start the gui demo on local touchscreen
}                   // NOTE: for QScreen, GUI_Demo MUST call Globalize_TVars(TVARS);

void main( void )
{              // comment in the correct version for your wildcard:
      Ether_GUI_Web_Demo();
//    WiFi_GUI_Web_Demo();
}

// NOTES: place a call to Ether_Check_GUI(E_MODULENUM); in the application task
// that invokes the GUI Routine
// (for example, after the call to Service_GUI_Events on a QVGA Controller system).
// Add a call to Screen_Has_Changed(); in each GUI handler that changes screen appearance.
// QScreen Controllers require a call to Globalize_TVars(TVARS); in the init function.
// When building tasks, do NOT include the statement NEXT_TASK = TASKBASE;
//  because we want to allow multiple tasks to be built.
// Type WARM or COLD before re-running the program.
```

Appendix D: Forth Demo Program

Listing 1-21 Ether_WiFi_Demo.4th program listing.

```
\ this demonstration code is provided in source form.
\ Top level functions:
\ Ether_Web_Demo ( -- ) \ no input parameters; runs web pages, enables email, tunneling
\ WiFi_Web_Demo ( -- )\ no input parameters; runs web pages, enables email, tunneling
\ Ether_Monitor_Demo  ( modulenum -- ) \ runs ethernet interactive monitor;see comments

\ these can be called interactively (from QEDTerminal)
\ after Ether_Web_Demo or WiFi_Web_Demo is invoked:
\  Tunnel_Test       \ no input parameters, echoes data from Putty connection
\  Email_Test        \ no input parameters, returns error code.  Sends an email;
                  \ Note: YOU MUST EDIT THIS FUNCTION's HARD-CODED IP ADDRESS and
                  \ the hostname_str, sender_str, and recipient_str BEFORE CALLING!
```

\ Make sure to edit the E_MODULENUM constant to match your hardware jumper settings.

```
\ ******************* SET MEMORY MAP HERE (IF IT'S NOT ALREADY SET) ****************


\ ************** USEFUL CONSTANTS **************


HEX
400 CONSTANT TASK_SIZE     \ 0x400 = decimal 1024 = 1Kbyte = task size

0D CONSTANT CR_ASCII    \ ascii carriage return
0D0A CONSTANT CRLF_ASCII   \ this is the standard eol sequence for tcp/ip services.

DECIMAL     \ rest of file is in decimal base
25 CONSTANT SMTP_PORT      \ email Simple Mail Transfer Protocol destination port


\ *************** IMPORTANT: SET MODULENUM TO MATCH HARDWARE JUMPERS ******************


FIND E_MODULENUM  \ don't redefine if already defined in a prior file
IFTRUE
XDROP
OTHERWISE
4 CONSTANT E_MODULENUM  \ ******** SET THIS TO MATCH HARDWARE JUMPERS J1 AND J2! *****
ENDIFTRUE
\ This specifies the modulenum for the high level functions in this file
\ except for Ether_Monitor_Demo (see its comments).


\ If EtherSmart is installed on module bus 0, E_MODULENUM = 0, 1, 2, or 3
\ If EtherSmart is installed on module bus 1, E_MODULENUM = 4, 5, 6, or 7
\ That is, the bus specifies the top bit of a 3-bit modulenum,
\ and [J2, J1] specifies the remaining 2 bits of the modulenum.
\ Example: On module bus 0, if neither jumper cap is installed: E_MODULENUM = 0
\ Example: On module bus 0, if both jumper caps are installed: E_MODULENUM = 3
\ Example: On module bus 1, if J2 is installed but J1 is not: E_MODULENUM = 6



\ ************** SETUP ETHERNET TASK TO RUN WEB AND TUNNELING SERVICES **************


TASK_SIZE V.INSTANCE:     ether_control_task  \ 1 Kbyte per task area

: Ether_Task_Setup_Default    ( -- )
```

```
\ performs full initialization of the ether_info struct and mailboxes for the
\ modulenum specified by E_MODULENUM, and
\ builds and activates an ethernet control task to service the EtherSmart Wildcard
\      (STATUS) NEXT.TASK ! \ empty the task loop; comment out if other tasks are allowed
     ether_control_task DROP E_MODULENUM     ( taskbase_addr\modulenum -- )
     Ether_Task_Setup                               ( -- )
;

: WiFi_Task_Setup_Default    ( -- )
\ same as Ether_Task_Setup_Default, but also marks module as wifi
\ performs full initialization of the ether_info struct and mailboxes for the
\ modulenum specified by E_MODULENUM, and
\ builds and activates an ethernet control task to service the WiFi Wildcard
\      (STATUS) NEXT.TASK ! \ empty the task loop; comment out if other tasks are allowed
      ether_control_task DROP E_MODULENUM   ( taskbase_addr\modulenum -- )
      WiFi_Task_Setup
;



\ ***************** INTERACTIVE CONFIGURATION AND REPORTING ************

: Ether_Set_Defaults ( -- error )
\ works for xport or wiport (ethernet/wifi)
\ call this AFTER calling Ether_Web_Demo or Ether_Task_Setup_Default  or
\ WiFi_Web_Demo or WiFi_Task_Setup_Default.
\ gets assigned via DHCP (Dynamic Host Configuration Protocol) by the LAN’s gateway.
\ see user guide for more information.
   CR .” Setting defaults...” CR
   E_MODULENUM Ether_XPort_Defaualts \ works for xport or wiport (ethersmart/wifi)
   E_MODULENUM Ether_Await_Response DROP \ error code is in lsword
;

: Ether_Set_Local_IP ( my_ip1\my_ip2\my\my_ip4 -- )
\ works for xport or wiport (ethernet/wifi)
\ call this AFTER calling Ether_Web_Demo or Ether_Task_Setup_Default  or
\ WiFi_Web_Demo or WiFi_Task_Setup_Default.
\ sets the IP address of the EtherSmart Wildcard specified by E_MODULENUM as:
\   ip1.ip2.ip3.ip4
\ For example, to set the IP address to 10.0.1.22, pass to this function the parameters:
\   10 0 1 22
\ returns error code
\ NOTES: type DECIMAL at the monitor before invoking this function interactively!
\ NOTE: assigning a nonzero IP address disables DHCP!
   CR .” Setting local IP address...” CR
   E_MODULENUM Ether_Local_IP       ( -- )
   E_MODULENUM Ether_XPort_Update
   E_MODULENUM Ether_Await_Response DROP \ error code is in lsword
;

: Ether_IP_Report ( -- )
\ works for xport or wiport (ethernet/wifi)
\ call this AFTER calling Ether_Web_Demo or Ether_Task_Setup_Default  or
\ WiFi_Web_Demo or WiFi_Task_Setup_Default.
\ takes 7 seconds to execute, so be patient.
\ Report is of the form:
\ IP 010.000.001.019 GW 010.000.001.022 Mask 255.255.255.000
\ which summarizes the IP address, gateway address, and netmask, respectively.
   E_MODULENUM Ether_IP_Info_Report
;

: Ether_Ping ( my_ip1\my_ip2\my\my_ip4 -- )
\ works for xport or wiport (ethernet/wifi)
\ call this AFTER calling Ether_Web_Demo or Ether_Task_Setup_Default  or
\ WiFi_Web_Demo or WiFi_Task_Setup_Default.
\ on error, prints “ Couldn’t enter monitor mode!” or “ No response from remote”.
\ takes thirteen seconds to execute, so be patient.
\ Report is of the form (summarizes response time from specified remote host):
```

```
\ Seq 001 time 10ms
\ Seq 002 time 10ms
\ Seq 003 time 10ms
\ Seq 004 time 10ms
\ Seq 005 time 10ms
\ Seq 006 time 10ms
\ NOTES: type DECIMAL at the monitor before invoking this function interactively!
   E_MODULENUM Ether_Ping_Report
;

\ ***************** REVECTORED SERIAL VIA TCP/IP ************

\ simplest approach:
\ simply store the correct modulenum into the global var named ether_revector_module,
\ then call Ether_Serial_Revector
\ This will run the default (startup) task through the specified ethersmart wildcard.
\ Use Putty “rlogin” to connect to the local port (typically = 80)
\ at the specified local IP address, and you’re talking to the QED monitor via ethernet.
\ To revert to standard serial operation via QED Term, type COLD to revert to standard
\ serial, then from QEDTerm type RESTORE to bring back access to all compiled routines,
\ then continue communications using QEDTerm.

\ If you want to maintain serial communications via QEDTerm with the default task
\ while running a separate task with I/O revectored via the Ethersmart Wildcard,
\ then build and activate a task using Ether_Monitor as the activation routine.
\ Use Putty “rlogin” to connect to the local port (typically = 80)
\ at the specified local IP address, and you’re talking to the task via ethernet.


TASK_SIZE V.INSTANCE:     ether_montask  \ 1 Kbyte per task area

: Ether_Monitor_Demo  ( modulenum -- )
\ works for xport or wiport (ethersmart or wifi)
\ builds and activates a monitor task for the specified module.
\ this function expects the modulenum as an input parameter so that
\ you can run 2 installed Ethersmart wildcards at once:
\ one wildcard can run the standard services (web, email, tunneling)
\ while the other one (running Ether_Monitor_Demo) is used to download code
\ and control program development and debugging.
\ To do this, first call Ether_Web_Demo or WiFi_Web_Demo
\ (which runs services on the E_MODULENUM wildcard),
\ then call this Ether_Monitor_Demo function with a different modulenum.
   1 NEEDED             ( modulenum--) \ make sure the input param is present
   ether_revector_module !    ( -- )
   RELEASE.ALWAYS SERIAL.ACCESS !      \ ensure lots of PAUSEs in Forth task
\  (STATUS) NEXT.TASK ! \ empty the task loop; comment out if other tasks are allowed
   0\0 0\0 0\0 ether_montask BUILD.STANDARD.TASK
   CFA.FOR Ether_Monitor ether_montask ACTIVATE
;



\ ****************** SERIAL TUNNELING TEST **************

: Tunnel_Test  ( -- )
\ works for xport or wiport (ethernet/wifi)
\ call this AFTER calling Ether_Task_Setup_Default  or WiFi_Task_Setup_Default.
\ Waits for incoming non-web connection on this module’s IP address on port 80
\ (the default local port), and examines the linefeed-delimited first line to see
\ if it starts with GET; if not, it accepts it as a passive non-web connection.
\ So: don’t type GET as the first characters of the connection (you’ll confuse the system).
\ The easiest way to open a connection is via Putty, using the “raw” mode.
\ Note: If you use “telnet” mode, you’ll see some garbage characters in the first line;
\ these are control bytes with the msbit set that are accepted by the connection manager.
   E_MODULENUM
   20000          \ use a 20 second timeout for testing; sets delay til function exit
   0 0 0             \ initial values for &error &num_received &numlines locals
   LOCALS{ &error &num_received &numlines &timeout &module }
   CR ." Waiting for connection (type a carriage return from QEDTerm to abort)..."
   CR ." (Suggestion: To connect, use Putty in 'raw' mode, specify IP address, port 80,"
   CR ."  and type a carriage return once the session window opens."
   CR
   BEGIN
      PAUSE.ON.KEY   \ abort on CR, pause/resume on other keys
      &module Ether_Check_Response 2DROP  ( -- ) \ keep mailbox clean, ignore messages
      &module Ether_Passive_Non_Web_Connection  ( tunneling_connection? -- )
   UNTIL
   CR ." An incoming connection has been accepted;"
   CR ." each incoming line will be printed to the serial terminal, and a"
   CR ." response prompt will be sent to the ethernet terminal."
   CR ." To exit this routine, wait 20 seconds without typing a line."
   CR
      \ the first line is in HTTP_Inbuf (because webserver had to check its identity)...
   &module HTTP_Inbuf LCOUNT TYPE   \ type first line & cr to local serial terminal
   BEGIN                  \ loop and echo additional lines til timeout...
      &module Ether_Inbuf &module Ether_Inbufsize CR_ASCII TRUE TRUE &timeout &module
            ( xlbuf\maxchars\eol\discard.alt.eol?\no.msbitset?\timeout_msec\module--)
      Ether_Get_Line             ( -- ) \ get 1 line into lcounted ether_inbuf
      &module Ether_Await_Response  ( numbytes_rcvd\cmd&module-- )
      DROP TO &num_received
      &module Ether_Inbuf LCOUNT TYPE  \ type line & cr to local serial terminal
      &num_received
      IF " Line was received>" COUNT &timeout &module
                         ( xaddr\count\timeout_msec\module_number -- )
         Ether_Send_Buffer       ( -- ) \ send prompt
         &module Ether_Await_Response ( numbytes_sent\cmd&module--) \ wait for result
         2DROP
         &numlines 1+ TO &numlines  ( -- ) \ bump numlines
      ENDIF
      &num_received 0=           ( done? -- )
   UNTIL    \ until no chars rcvd due to timeout or connection close
   &module Ether_Disconnect_Flush      ( -- )   \ clean up
   &module Ether_Await_Response 2DROP  ( -- ) \ synchronize to result mailbox before exit
   CR ." Connection terminated." CR
;


\ ********************* EMAIL  TEST ***************************

\ Use CREATE for relocatable strings. CFA.FOR HOSTNAME$ returns string address.
CREATE hostname_str " wildcard.yourdomain.com" XDROP  \ drop string xaddr
CREATE sender_str   " niceguy@yourdomain.com" XDROP   \ drop string xaddr
CREATE recipient_str " notso_niceguy@yourdomain.com" XDROP  \ drop string xaddr

CREATE email_body_str   \ use cfa.for email_body_str to obtain the longbuffer xaddress
DP  -1 CRLF_ASCII ASCII }     ( xpointer.to.result.area\max.chars\EOL\delim--)
LPARSE
Subject: EtherSmart/WiFi Email Test
This is a test email from the EtherSmart/WiFi Wildcard.
Emails can of course have multiple lines... }
XDROP \ drop string xaddr; use cfa.for <name>


: Email_Test   ( -- error )
\ works for xport or wiport (ethernet/wifi)
\ call this AFTER calling Ether_Task_Setup_Default  or WiFi_Task_Setup_Default.
\ smtp email test routine.
\ Note: YOU MUST EDIT THIS FUNCTION'S HARD-CODED IP ADDRESS
\  AND the hostname_str, sender_str, and recipient_str BEFORE CALLING!
\  Otherwise, it will not work on your system.
\ error= 0 on success, or error_no_response = 0x10, or the smtp 3-digit decimal error code
   E_MODULENUM
   LOCALS{ &module }
   CFA.FOR email_body_str LCOUNT       ( xbody_string\cnt--) \ subject plus 2-line email
   CFA.FOR hostname_str COUNT
   CFA.FOR sender_str COUNT
   CFA.FOR recipient_str COUNT
   &module Ether_Inbuf     \ scratchpad xbuffer
   10 10 10 1              \ IMPORTANT: ENTER YOUR GATEWAY'S IP ADDRESS HERE!
   SMTP_PORT             \ port 25 = standard mailserver destination port
   10000              \ use a 10 second timeout (10,000 msec) for testing
   &module
   ( email_body_xaddr\ecnt\hostname_xaddr\hcnt\sender_xaddr\scnt\rcvr_xaddr\rcnt
   \  \scratchpad_xbuf\dest_ip1\ip2\ip3\ip4\dest_port\timeout_msec\modulenum -- )
   Ether_Send_Email           ( -- )
   &module Ether_Await_Response  ( -- d.mailbox_contents ) \ error code is in lsword
   DROP                     ( -- error )
;



\ *********************** WEBSERVER DEMO (Ethersmart Web Pages) ***************


\ before sending this file (in the loader that includes this file),
\ include the s-record version of the html pages and the image(s) as:
\ #include "Resources\image_data.txt"
\ see the demo includer file


\ before sending this file (in the loader that includes this file),
\ include the header file that defines the base xaddress and 32-bit size for each resource:
\ #include "Resources\image_headers.4th"

DECIMAL  \ confirm compilation base in case the include files changed them

\ each file name is concatenated via the _ character with its extension,
\ and 32-bit constants are created with the _XADDR and _SIZE endings.
\ NOTE: Make each filename C-compatible, using only letters, _ (underscore), and
\  non-leading numerals in the filename.
\ In this demo program the resources comprise the image file mosaic_logo.png,
\ and the html files form_page.html, home_body.html, and opera_config.html.
\ The following 32-bit constants are declared in the image_headers.txt file:
\ FORM_PAGE_HTML_XADDR
\ FORM_PAGE_HTML_SIZE
\ HOME_BODY_HTML_XADDR
\ HOME_BODY_HTML_SIZE
\ MOSAIC_LOGO_PNG_XADDR
\ MOSAIC_LOGO_PNG_SIZE
\ OPERA_CONFIG_HTML_XADDR
\ OPERA_CONFIG_HTML_SIZE


\ note: in form_page_str, we mark one of the radio buttons as checked; otherwise,
\ the browser can skip the field entirely, then we would have to match the fieldnames!


CREATE form_response_str   \ cfa.for form_response_str to obtain the longbuffer xaddress
DP  -1 CRLF_ASCII ASCII `      ( xpointer.to.result.area\max.chars\EOL\delim--)
LPARSE
<html><head><title>Form Response</title></head><body><H3>Thanks for your response!</H3><p>
<p>It must be great to be a `
XDROP          ( -- ) \ drop x$addr, use cfa.for to get relocatable runtime xl$addr


: Home_Page ( modulenum -- )
\ this is the handler function for the /index.html and the / URLs.
\ sends the http header with dynamic (because of changing time stamp) text/html
\ content type,  followed by home_body_str text and the elapsed seconds since the
\ system was initialized.
   BASE @ DECIMAL    \ display to web page in decimal base
   LOCALS{ &prior_base &module }
   &module HTTP_Outbuf &module HTTP_Outbufsize HTTP_Put_Header \ http header->outbuf
   &module HTTP_Outbuf &module HTTP_Outbufsize
```

```
    -1 HTTP_TEXT_HTML_CONTENT        ( xlbuf\maxbufsize\dynamic?\content--)
    HTTP_Put_Content_Type            \ dynamic text/html content ->outbuf;header is done
    &module HTTP_Outbuf &module      ( xbuf\module--)
    HTTP_Send_LBuffer DROP           ( -- ) \ send out header, drop #bytes_sent
    HOME_BODY_HTML_XADDR HOME_BODY_HTML_SIZE D>S &module  ( xlstring\count\module_num--)
    HTTP_Send_Buffer DROP            ( -- ) \ send direct from str buf, drop #bytes_sent
    &module HTTP_Outbuf OFF          \ reset lcount to start fresh in buffer
    READ.ELAPSED.SECONDS ROT DROP       ( ud#sec-- ) \ drop #msec!
    <# #S #>                   ( xaddr\cnt -- ) \ convert time to ascii string
    &module                    ( xstring_to_add\cnt_to_add\module--)
    HTTP_Outbuf_Cat                  ( -- ) \ add line & crlf to longstring, update lcnt
    " seconds since system initialization.</body></html>"
    COUNT &module  ( xstring_to_add\cnt_to_add\module_num--)
    HTTP_Outbuf_Cat                  ( -- ) \ add line & crlf to longstring, update lcnt
    &module HTTP_Outbuf &module      ( xbuf\module--)
    HTTP_Send_LBuffer DROP           ( -- ) \ send final line, drop #bytes_sent
    &prior_base BASE !               ( -- ) \ restore prior numeric base
;


: Opera_Config_Page  ( modulenum -- )
\ this is the handler function for the /opera_configuration.html URL.
\ sends the http header with static text/html content type,
\ followed by opera_config_str text
   LOCALS{ &module }
   &module HTTP_Outbuf &module HTTP_Outbufsize HTTP_Put_Header \ http header->outbuf
   &module HTTP_Outbuf &module HTTP_Outbufsize
   0 HTTP_TEXT_HTML_CONTENT          ( output_xlbuf\max_bufsize\dynamic?\content_id -- )
   HTTP_Put_Content_Type             \ static text/html content ->outbuf;header is done
   &module HTTP_Outbuf &module       ( xbuf\module--)
   HTTP_Send_LBuffer DROP            ( -- ) \ send out header, drop #bytes_sent
   OPERA_CONFIG_HTML_XADDR OPERA_CONFIG_HTML_SIZE D>S &module ( xlstring\count\modulenum--)
   HTTP_Send_Buffer DROP             ( -- ) \ send direct from str buf, drop #bytes_sent
;

: Form_Entry_Page ( modulenum -- )
\ responds to url= /form_entry.html
\ places FORM_TEXT into the buffer as a longstring after the http header.
\ we send directly from compiled string in flash to avoid copy into ether_out buffer;
\ this method is ideal for serving static text.
   LOCALS{ &module }
   &module HTTP_Outbuf  &module HTTP_Outbufsize 3DUP
   HTTP_Put_Header                   ( xbuffer\bufsize -- )
   0 HTTP_TEXT_HTML_CONTENT          ( output_xlbuf\max_bufsize\dynamic?\content_id -- )
   HTTP_Put_Content_Type             ( -- ) \ static text/html content
   &module HTTP_Outbuf &module
   HTTP_Send_LBuffer DROP            ( -- ) \ drop #bytes_sent; header has been sent
   FORM_PAGE_HTML_XADDR FORM_PAGE_HTML_SIZE D>S &module     ( xbuf\count\module--)
   HTTP_Send_Buffer DROP             ( -- ) \ drop #bytes_sent
;



: Form_Response_Page ( modulenum -- )
\ this is the handler function that responds to url= /form_response.cgi
\ incoming request from browser looks like this:
\ GET /form_response.cgi?classification=<man/woman/child>&name_id=<namestring>
\ &color=<red/blue/yellow/green>
\ We respond:
\ It must be great to be a <man/woman/child/person> with a name like <namestring/yours>.
\ It's good to hear that <red/blue/yellow/green/rainbow> is your favorite color.
   DUP HTTP_Outbuf
   LOCALS{ x&lstring_buf &module | x&next_value_ptr &next_value_count }
   x&lstring_buf  &module HTTP_Outbufsize DUP>R
   HTTP_Put_Header                   ( -- )
   x&lstring_buf  R> 0 HTTP_TEXT_HTML_CONTENT   ( xlbuf\max_bufsize\dynamic?\content_id--)
   HTTP_Put_Content_Type             ( -- ) \ static text/html content
   x&lstring_buf &module
   HTTP_Send_LBuffer DROP           ( -- ) \ header has been sent; drop #bytes_sent
   CFA.FOR form-response_str &module   ( xbuf\module--)
   HTTP_Send_LBuffer DROP           ( -- ) \ send static text, drop #bytes_sent
   x&lstring_buf OFF              \ reset lbuffer count
   &module HTTP_Value_Ptr &module HTTP_Value_Count DUP 0=   ( class_xaddr\cnt\null?--)
   IF 3DROP " person" COUNT         ( xstring_to_add\cnt_to_add--) \ substitution
   ENDIF
   &module HTTP_Outbuf_Cat       ( -- ) \ add line & crlf to longstring, update lcnt
   "  with a name like " COUNT &module HTTP_Outbuf_Cat   ( -- ) \ add line, update lcnt
   &module HTTP_To_Next_Field DROP     ( -- ) \ go to name_id field, drop #chars_advanced
   &module HTTP_Value_Ptr &module HTTP_Value_Count DUP   ( nameID_xaddr\cnt\present?--)
   &module HTTP_To_Next_Field DROP     ( nameID_xaddr\cnt\present?--) \ go to color field
   &module HTTP_Value_Ptr TO x&next_value_ptr   \ get next field params before we unescape!
   &module HTTP_Value_Count TO &next_value_count   ( nameID_xaddr\cnt\present?--)
   IF 3DUP HTTP_Plus_To_Space       ( nameID_xaddr\cnt--) \ get rid of +'s in name text
      >R XDUP R>                 ( xaddr\xaddr\cnt -- )
      HTTP_Unescape              ( xaddr\revised_cnt -- ) \ get rid of special chars
   ELSE 3DROP " yours" COUNT        ( xstring_to_add\cnt_to_add--) \ "name like yours"
   ENDIF
   &module HTTP_Outbuf_Cat       ( -- ) \ add line & crlf to longstring, update lcnt
   " . It's good to hear that " COUNT  ( xaddr\cnt-- )
   &module HTTP_Outbuf_Cat       ( -- ) \ add line & crlf to longstring, update lcnt
   x&next_value_ptr &next_value_count DUP 0= ( color_xaddr\cnt\empty--)
   IF 3DROP " rainbow" COUNT        ( xstring_to_add\cnt_to_add--) \ substitution
   ENDIF
   &module HTTP_Outbuf_Cat       ( -- ) \ add line & crlf to longstring, update lcnt
   "  is your favorite color!</body></html>" COUNT
   &module HTTP_Outbuf_Cat       ( -- ) \ add line & crlf to longstring, update lcnt
   x&lstring_buf &module
   HTTP_Send_LBuffer DROP           ( -- ) \ page has been sent; drop #bytes_sent
;


: Logo_Response_Page ( module -- )
\ this is the handler function that responds to url= /logo_response.png
\ assumes that logo.s2 has been loaded into memory. serves it as static image.
   DUP HTTP_Outbuf
   LOCALS{ x&lstring_buf &module }
   x&lstring_buf &module HTTP_Outbufsize DUP>R
   HTTP_Put_Header                  ( -- )
   x&lstring_buf  R> 0 HTTP_IMAGE_PNG_CONTENT ( xlbuf\max_bufsize\dynamic?\content_id--)
   HTTP_Put_Content_Type            ( -- ) \ static png image content
   x&lstring_buf LCOUNT           ( xaddr_buf1\count1--)
   MOSAIC_LOGO_PNG_XADDR MOSAIC_LOGO_PNG_SIZE D>S
   &module                        ( xbuf1\cnt1\xbuf2\count2\module--)
   HTTP_Send_2Buffers DROP             ( -- ) \ send header&image, drop numbytes_sent
;


: Web_Handler_Installation ( modulenum -- error? )
\ call this after Ether_Task_Setup_Default
\ point browser to raw ip or to ip/index.html to see the home web page.
\ urls are case sensitive. any other url's serve out: page not found.
\ returns nonzero error if too many handlers were added
\ (limited by AUTOSERVE_DEFAULT_ROWS passed to ether_init)
   0 LOCALS{ &error &module }
   " /" COUNT CFA.FOR Home_Page &module   ( url_xaddr\count\handler_xcfa\mod--)
   HTTP_Add_Handler &error OR TO &error   ( -- )
   " /index.html" COUNT CFA.FOR Home_Page &module ( url_xaddr\count\xcfa\mod--)
   HTTP_Add_Handler &error OR TO &error   ( -- )
   " /opera_configuration.html" COUNT
   CFA.FOR Opera_Config_Page &module ( xaddr\count\xcfa\mod--)
   HTTP_Add_Handler &error OR TO &error   ( -- )
   " /form_entry.html" COUNT CFA.FOR Form_Entry_Page &module ( xaddr\cnt\xcfa\mod--)
   HTTP_Add_Handler &error OR TO &error   ( -- )
   " /form_response.cgi" COUNT CFA.FOR Form_Response_Page &module ( xadr\cnt\xcfa\mod--)
   HTTP_Add_Handler &error OR TO &error   ( -- )
   " /logo_response.png" COUNT CFA.FOR Logo_Response_Page &module ( xadr\cnt\xcfa\mod--)
   HTTP_Add_Handler &error OR TO &error   ( -- )
   INIT.ELAPSED.TIME    \ start at zero so home page reports correct elapsed time
   &error                          ( error? -- )
;


: Ether_Web_Demo  ( -- )
\ for EtherSmart Wildcard: point browser to raw ip or to
\  ip/index.html to see the home web page.  urls are case sensitive.
\ any other url's serve out: page not found.
\ returns nonzero error if too many handlers were added
\ (limited by AUTOSERVE_DEFAULT_ROWS passed to ether_init)
   Ether_Task_Setup_Default
   E_MODULENUM Web_Handler_Installation DROP ( -- ) \ ignore too_many_handlers? flag
;

: WiFi_Web_Demo  ( -- )
\ for WiFi Wildcard: point browser to raw ip or to
\  ip/index.html to see the home web page.  urls are case sensitive.
\ any other url's serve out: page not found.
\ returns nonzero error if too many handlers were added
\ (limited by AUTOSERVE_DEFAULT_ROWS passed to ether_init)
   Ether_Task_Setup_Default
   E_MODULENUM Web_Handler_Installation DROP ( -- ) \ ignore too_many_handlers? flag
;
```

Appendix E: Forth Remote Front Panel Demo Program

Listing 1-22 Ether_GUI_Demo.4th program listing.

```
\ this demonstration code is provided in source form.
\ Top level functions:
\  Ether_GUI_Web_Demo ( -- ) \ no input parameters; runs gui and web page on EtherSmart WC
\  WiFi_GUI_Web_Demo ( -- ) \ no input parameters; runs gui and web page on WiFi WC

\ ******************* SET MEMORY MAP HERE (IF IT'S NOT ALREADY SET) ****************


\ ************** USEFUL CONSTANTS **************

HEX      \ rest of file in hex base
400 CONSTANT TASK_SIZE     \ 0x400 = decimal 1024 = 1Kbyte = task size

\ 0D0A CONSTANT CRLF_ASCII \ this is the standard eol sequence for tcp/ip services.

\ *************** IMPORTANT: SET MODULENUM TO MATCH HARDWARE JUMPERS ******************

\ recall: modulenum 0 is reserved on the QScreen!

FIND E_MODULENUM  \ don't redefine if already defined in a prior file
IFTRUE
XDROP
OTHERWISE
4 CONSTANT E_MODULENUM  \ ******** SET THIS TO MATCH HARDWARE JUMPERS J1 AND J2! *****
ENDIFTRUE
\ This specifies the modulenum for the high level functions in this file
\ except for Ether_Monitor_Demo (see its comments).

\ If EtherSmart/WiFi is installed on module bus 0, E_MODULENUM = 0, 1, 2, or 3
\ If EtherSmart/WiFi is installed on module bus 1, E_MODULENUM = 4, 5, 6, or 7
\ That is, the bus specifies the top bit of a 3-bit modulenum,
\ and [J2, J1] specifies the remaining 2 bits of the modulenum.
\ Example: On module bus 0, if neither jumper cap is installed: E_MODULENUM = 0
\ Example: On module bus 0, if both jumper caps are installed: E_MODULENUM = 3
\ Example: On module bus 1, if J2 is installed but J1 is not: E_MODULENUM = 6


\ ************** SETUP ETHERNET TASK TO RUN WEB AND TUNNELING SERVICES **************


TASK_SIZE V.INSTANCE:     ether_control_task  \ 1 Kbyte per task area

: Ether_Task_Setup_Default    ( -- )
\ performs full initialization of the ether_info struct and mailboxes for the
\ specified EtherSmart modulenum, and
\ builds and activates an ethernet control task to service the xport
\  (STATUS) NEXT.TASK ! \ empty the task loop; comment out if other tasks are allowed
   ether_control_task DROP E_MODULENUM ( taskbase_addr\modulenum -- )
   Ether_Task_Setup              ( -- )
;

: WiFi_Task_Setup_Default    ( -- )
\ performs full initialization of the ether_info struct and mailboxes for the
\ specified WiFi modulenum, and
\ builds and activates an ethernet control task to service the xport
\  (STATUS) NEXT.TASK ! \ empty the task loop; comment out if other tasks are allowed
   ether_control_task DROP E_MODULENUM ( taskbase_addr\modulenum -- )
   Ether_Task_Setup              ( -- )
;
```

```
\ *********************** WEBSERVER ***************************


\ REMOTE FRONT PANEL (QVGA benchmark: 11 seconds from mouse click to new screen image)

FIND }$      \ define ram location
IFTRUE       \ if V6.xx kernel...
   XDROP
   8000 14
OTHERWISE
   0 2       \ V4.xx..
ENDIFTRUE
XCONSTANT BITMAP_BUFFER
8000 CONSTANT BITMAP_MAXBYTES \ 32K max, plenty for monochrome displays


1 CONSTANT BITMAP_SCREEN_FORMAT

\ the following is now created using the Image Converter program which creates 2 constants:
\ GUI_RESPONSE_TEXT_HTML_XADDR
\ GUI_RESPONSE_TEXT_HTML_SIZE

\ CREATE GUI_RESPONSE_TEXT \ cfa.for GUI_RESPONSE_TEXT to obtain lbuffer xaddr
\ DP  -1 CRLF_ASCII ASCII $      ( xpointer.to.result.area\max.chars\EOL\delim--)
\ LPARSE
\ <html><head><title>EtherSmart/GUI Remote Front Panel</title></head>
\ <body>
\ <H3>EtherSmart/GUI Remote Front Panel</H3><p>
\ <H4>Click on the screen image to operate the touchscreen remotely</H3><p><p>
\ <a href="/gui_response.cgi"><img src="/screen_image.bmp" ismap></a>
\ </body></html>
\ $      \ end of gui_response_text string
\ XDROP           ( -- ) \ drop x$addr, use cfa.for to get relocatable runtime xl$addr
\ make sure to declare gui_response.cgi as Cache-Control: no-cache (dynamic content).
\ Notes: this works very nicely with Opera, the recommended browser.


: Screen_Image_Response ( module -- )
\ handler for /screen_image.bmp url. serves the screen image.
\ assumes that Simulated_Touch_To_Image has run.
\ xbuffer contains 32-bit count (NOTE!) followed by image data.
\ marks as no-cache (always reloads).
\ NOTE: this function is NOT a gui handler; it should be posted by http_add_handler.
   DUP HTTP_Outbuf
   LOCALS{ x&lstring_buf &module }
   x&lstring_buf &module HTTP_Outbufsize DUP>R
   HTTP_Put_Header                        ( -- )
   x&lstring_buf  R> -1 HTTP_IMAGE_BITMAP_CONTENT  ( xlbuf\maxbufsize\dynamic\content_id--)
   HTTP_Put_Content_Type            ( -- ) \ dynamic bitmap image content
   x&lstring_buf LCOUNT          ( xbuf1\cnt1--) \ http header in xbuf1
   BITMAP_BUFFER 2XN+ LCOUNT &module   ( xbuf1\cnt1\xbuf2\cnt2\mod--) \ 2xn+: 32-bit cnt!
   HTTP_Send_2Buffers DROP          ( -- ) \ drop numbytes_sent
;

: GUI_Response ( module -- )
\ a clickable ismap version of the screen image that runs the gui toolkit simulated touch.
\ url = /gui_response.cgi
\ this routine is called via ether_check_gui which runs in the application task
\ that runs Service_GUI_Events.  it is posted using http_gui_add_handler.
   DUP HTTP_Outbuf
   LOCALS{ x&lstring_buf &module }
   &module HTTP_Imagemap SWAP          ( x\y-- ) \ parses query response= ?x,y
   DUP 0<                           \ returns -1\-1 if ?x,y is not present
   IF 2DROP              ( -- ) \ invalid coordinates: do nothing (should never happen)
   ELSE                          ( x\y-- )
      X>R
      BITMAP_BUFFER BITMAP_MAXBYTES BITMAP_SCREEN_FORMAT
      XR>                        ( xlbuffer\maxbytes\format\x\y--)
      Simulated_Touch_To_Image         ( error -- ) \ if 0, image has changed
      DROP                       ( --) \ we always try to re-serve image
   ENDIF
   x&lstring_buf &module HTTP_Outbufsize DUP>R
   HTTP_Put_Header               ( -- )
   x&lstring_buf  R> 0 HTTP_TEXT_HTML_CONTENT ( xlbuf\maxsize\dynamic\content_id--)
   HTTP_Put_Content_Type         ( -- ) \ static html text content
   x&lstring_buf LCOUNT
   GUI_RESPONSE_TEXT_HTML_XADDR GUI_RESPONSE_TEXT_HTML_SIZE D>S
   &module                    ( xaddr_buf1\count1\xaddr_buf2\count2\module--)
   HTTP_GUI_Send_2Buffers        ( -- ) \ numbytes_sent is not returned
;

: Init_Screen_Image  ( -- error )
   BITMAP_BUFFER BITMAP_MAXBYTES BITMAP_SCREEN_FORMAT ( xlbuf\maxbytes\format--)
   Screen_To_Image         ( error -- ) \ if 0, image has been created
;

: REMOTE_PANEL_START ( module -- )
\ serves an initial clickable ismap version of the screen image.
\ url = /remote_panel_start.html
\ puts initial version of screen into bitmap_buffer.
\ this routine is called via ether_check_gui which runs in the application task
\ that runs Service_GUI_Events.  it is posted using http_gui_add_handler
   DUP HTTP_Outbuf
   LOCALS{ x&lstring_buf &module }
   x&lstring_buf &module HTTP_Outbufsize DUP>R  \ if new image, serve the page
   HTTP_Put_Header               ( -- )
   x&lstring_buf  R> 0 HTTP_TEXT_HTML_CONTENT ( xlbuf\maxsize\dynamic\content_id--)
   HTTP_Put_Content_Type         ( -- ) \ static html text content
   x&lstring_buf LCOUNT       ( xbuf1\cnt1--) \ http header
   Init_Screen_Image          ( xbuf1\cnt1\error -- ) \ if 0, image has been created
   DROP                    ( xbuf1\cnt1--) \ ignore error flag
   GUI_RESPONSE_TEXT_HTML_XADDR GUI_RESPONSE_TEXT_HTML_SIZE D>S ( xbuf1\cnt1\xbuf2\cnt2--)
   &module                    ( xaddr_buf1\count1\xaddr_buf2\count2\module--)
   HTTP_GUI_Send_2Buffers        ( -- ) \ numbytes_sent is not returned
;

: Install_GUI_Web_Handlers ( module -- error? )
\ call this after ETHER_Task_Setup_DEFAULT  point browser to raw ip or to
\  ip/index.html to see the home GUI web page = remote front panel.
\ urls are case sensitive.
\ any other url's serve out: page not found.
\ returns nonzero error if too many handlers were added
\ (limited by AUTOSERVE_DEFAULT_ROWS passed to ether_init)
\  Some of the gui web handlers in this example are handled by the application task.
   0 LOCALS{ &error &module }
   " /" COUNT CFA.FOR Remote_Panel_Start &module   ( url_xaddr\count\handler_xcfa\mod--)
   HTTP_Add_GUI_Handler TO &error   ( -- )
   " /index.html" COUNT CFA.FOR Remote_Panel_Start &module ( url_xaddr\count\xcfa\mod--)
   HTTP_Add_GUI_Handler &error OR TO &error  ( -- )
   " /screen_image.bmp" COUNT CFA.FOR Screen_Image_Response
   &module              ( xadr\cnt\xcfa\mod--)
   HTTP_Add_Handler &error OR TO &error   ( -- )
   " /gui_response.cgi" COUNT CFA.FOR GUI_Response
   &module              ( xadr\cnt\xcfa\mod--)
   HTTP_Add_GUI_Handler &error OR   ( error -- )
;


: Ether_GUI_Web_Demo ( -- error )
   Ether_Task_Setup_Default            ( -- ) \ init EtherSmart WC, start ethernet task
   E_MODULENUM Install_GUI_Web_Handlers DROP ( -- ) \ setup web; ignore error flag
   Gui_Demo ( -- ) \ start the gui demo on local touchscreen
;

: WiFi_GUI_Web_Demo ( -- error )
   WiFi_Task_Setup_Default            ( -- ) \ init EtherSmart WC, start ethernet task
   E_MODULENUM Install_GUI_Web_Handlers DROP ( -- ) \ setup web; ignore error flag
   Gui_Demo ( -- ) \ start the gui demo on local touchscreen
;

\ NOTES: place the following function call:
\  E_MODULENUM Ether_Check_GUI DROP
\ in the application task that invokes the GUI Routine
\ (for example, after the call to Service_GUI_Events on a QVGA Controller system).
\ Add a call to Screen_Has_Changed in each GUI handler that changes screen appearance.
\ QScreen Controllers require this in the init function:  TVARS Globalize_TVars
\ When building tasks, do NOT include the statement:   (STATUS) NEXT.TASK !
\  because we want to allow multiple tasks to be built.
\ Type WARM or COLD before re-running the program.
```

Appendix F: Browser Configuration

You'll use a web browser on your PC to interact with the web server running on the EtherSmart/WiFi Wildcard. Popular browsers include Microsoft Internet Explorer, Netscape based browsers such as Firefox and Mozilla, and other high quality free browsers such as Opera. All of these browsers work with the web demonstration program that comes with the EtherSmart Wildcard.

Additional considerations can limit the performance of some of these browsers if your application needs to serve out more complex web pages that require more than one TCP/IP connection per page. This can occur, for example, when mixed text and image data originating from the Wildcard are served out in a single web page.

The Lantronix hardware on the EtherSmart/WiFi Wildcard supports only one active connection at a time. However, the HTTP/1.1 standard (and consequently all browsers in their default configuration) expect the webserver to be able to host two simultaneous connections. A default-configured browser will try to open a second connection when two or more content types (for example, HTML text and a JPEG image) are present in a single web page. The second connection will typically be refused by the Lantronix hardware, causing an incomplete page load. The solution is to configure the browser to expect only one connection from the webserver.

This Appendix explains how to reconfigure Internet Explorer to work with any web page that the EtherSmart/WiFi Wildcard can serve out. For the best solution, though, consider downloading the free Opera browser and using it for all your interactions with the EtherSmart/WiFi Wildcard.

Using Opera Is Highly Recommended

To be able to browse mixed text and graphics pages from the EtherSmart Wildcard without modifying your default browser, go to www.opera.com and download the latest version of the Opera browser for Windows desktop machines. It's free, the download file is compact, and the install takes only a few seconds.

Simply go to www.opera.com and select "Download Opera", then double-click on the resulting file to install the browser on your desktop. It is very easy to configure Opera for the EtherSmart/WiFi webserver. Once Opera is installed, simply go to its Tools menu, and select:

Preferences->Advanced->Network->Max Connections Per Server

and enter 1 in the box. Now you're ready to use Opera with the EtherSmart/WiFi Wildcard dynamic webserver described above.

Reconfiguring the Internet Explorer Browser

The Lantronix hardware on the EtherSmart/WiFi Wildcard implements only one connection at a time. To serve mixed text and images or text and frames, or any mixed data types, HTTP/1.1 browsers expect the server to host up to two simultaneous connections. Internet explorer works well with the Lantronix device if you set the registry key **`MaxConnectionsPerServer`** to 1.

To perform this browser configuration on your PC running Windows XP, you can (after backing up your registry):

1. Click Start | Run.
2. Type `regedit` and click OK.
3. Expand the directories HKEY_CURRENT_USER, then Software, Microsoft, Windows, CurrentVersion.
4. Click on "Internet Settings" to view its contents.
5. Check the right-hand column of the regedit window for the following line (value):

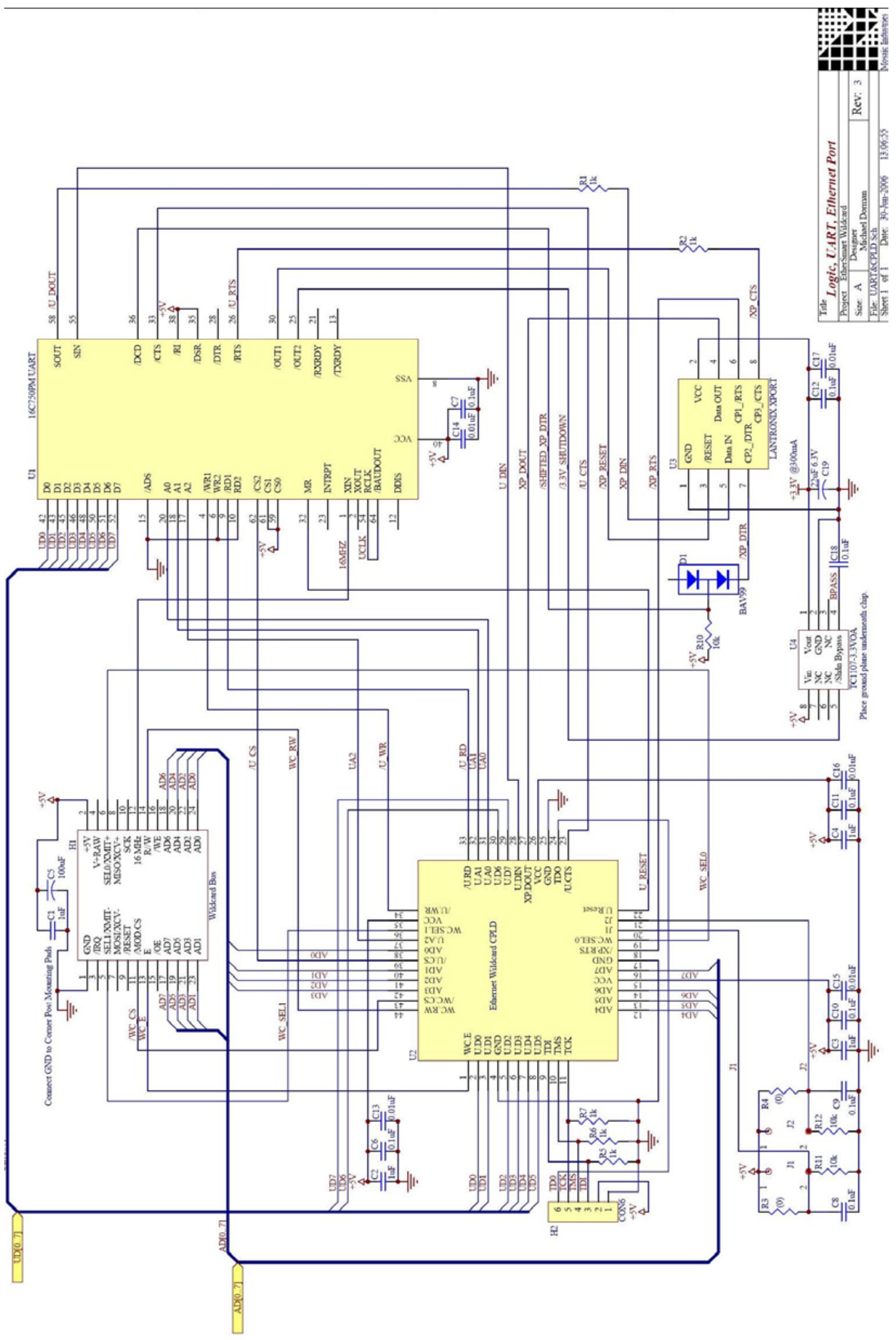
   `MaxConnectionsPerServer`

6. If present, right-click on the value (`MaxConnectionsPerServer`), select `Modify` from the drop-down menu, click Decimal, and set the Value data field to 1.
7. If the line (value) is not listed, right-click on the white region of Regedit's right-hand column, click `New`, and then click `DWORD Value`. Carefully type `MaxConnectionsPerServer` for the name of the new `DWORD Value` and press Enter. The new value should now appear in Regedit's right-hand column. Right-click the new value and click Modify. Click Decimal and set the Value Data field to 1, then click OK.

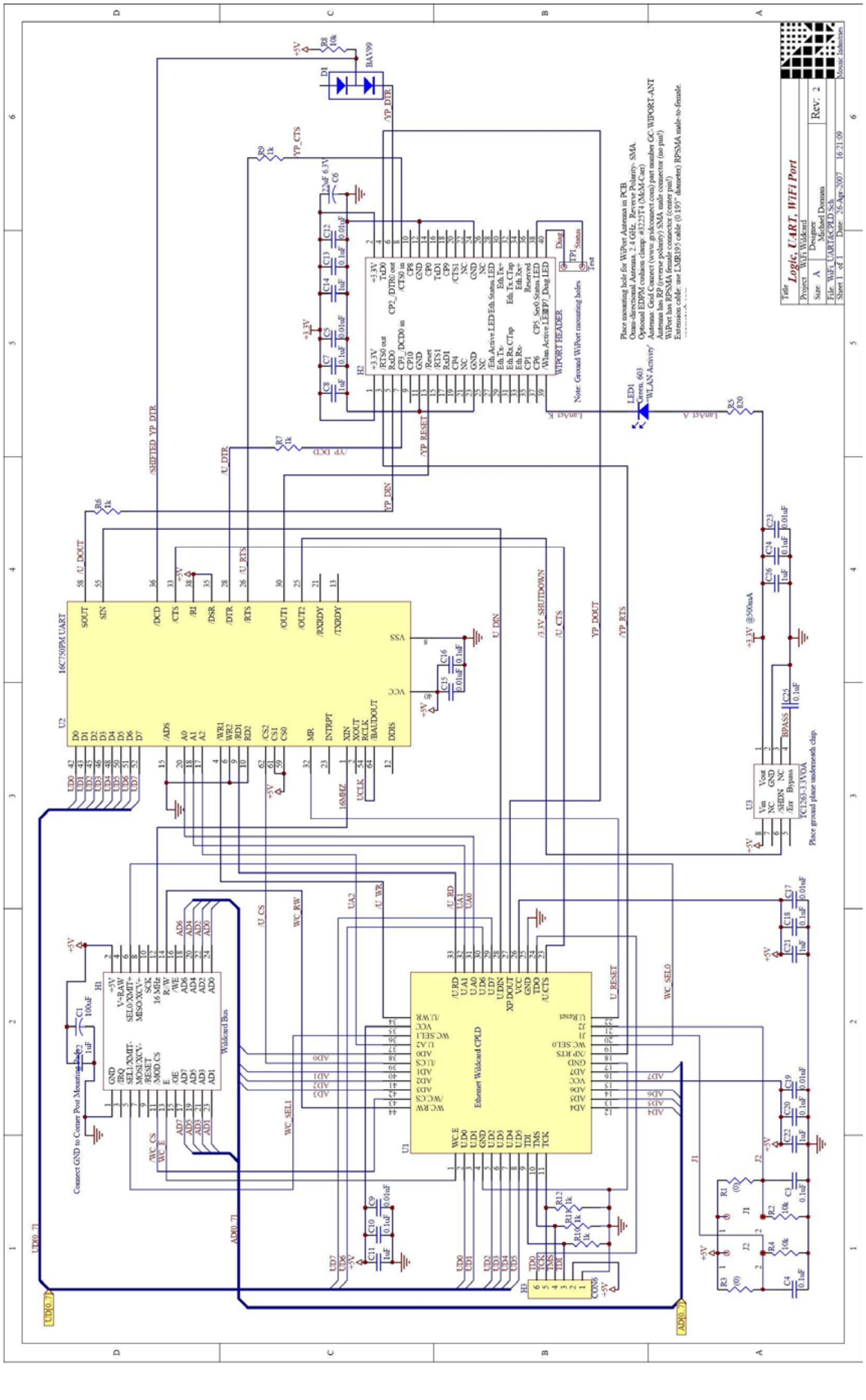
Firefox and Mozilla

Unfortunately, there is a low-level bug in the code base of the Netscape-based browsers such as Firefox and Mozilla that prevents reliable operation with mixed image type web pages (such as mixed text and images served from the Lantronix device). The Lantronix hardware does not increment its source port number with each exchanged TCP/IP packet, and Firefox and Mozilla rely on source port incrementing to properly order the received packets in time. The result is that some information is displayed in a skewed order by these browsers. This bug has been reported but has not been repaired. For simple single content type web pages such as those in the EtherSmart/WiFi demo program, any browser including Firefox and Mozilla works well. For more complex mixed data type web pages, Opera is recommended: download it for free from www.opera.com.

Appendix G: EtherSmart Hardware Schematics



Appendix H: WiFi Wildcard Hardware Schematics


Logic, UART, WiFi Port
Project: WiFi Wildcard
Designer: Michael Dorman
Rev: 2
Date: 26-Apr-2007 16:21:09
Mosaic Industries
16C750PM UART
Ethernet Wildcard CPLD
WIPORT HEADER
Wildcard Bus
TC1263-3.3VOA
Place ground plane underneath chip.
Note: Ground WiPort mounting holes
Connect GND to Corner Post Mounting Holes
Place mounting hole for WiPort Antenna in PCB
Omni-directional Antenna, 2.4 GHz, Reverse Polarity- SMA
Optional EDPM cushion clamp: #3225T4 (McM-Carr)
Antenna: Grid Connect (www.gridconnect.com) part number GC-WIPORT-ANT
Antenna has RP (reverse polarity) SMA male connector (no pin!)
WiPort has RPSMA female connector (center pin!)
Extension cable: use LMR195 cable (0.195" diameter) RPSMA male-to-female.