Glossary I

The EtherSmart/WiFi Wildcard Glossary

This Glossary provides detailed descriptions of the driver functions that control the EtherSmart and WiFi Wildcards. The EtherSmart Wildcard provides wired Ethernet connectivity, and the WiFi Wildcard provides wireless 802.11b/g connectivity. These Wildcards are implemented using similar hardware that enables a common software driver to support both. For each function, a prototype is provided to illustrate the function input and output parameters in both C and Forth formats, and a glossary entry describes the function's actions. These driver routines are available as pre-compiled libraries for the controller platforms from Mosaic Industries, for both C and Forth programmers. This Glossary contains the following:

- ⇨ *An introduction that presents an overview of the glossary entries and defines the terms used in this document;*
- ⇨ *A categorized list of all EtherSmart/WiFi Wildcard library functions;*
- ⇨ *The EtherSmart/WiFi Wildcard glossary; and,*
- ⇨ *A brief glossary of GUI Toolkit functions that help to implement a web-based "remote front panel".*

Introduction to the EtherSmart/WiFi Wildcard Glossary

The Form of the Glossary Definitions

Each entry in the Main Glossary of Library Functions includes a C prototype declaration, a Forth prototype declaration, and a detailed definition of what the function does. For constants, the prototype simply states the name of the routine. An example of a typical function declaration is as follows:

```
C:   uint Ether_Setup ( xaddr xbuffer_area_base, addr mailbox_base_addr, int modulenum )
4th: Ether_Setup ( xbuffer_area_base\ mailbox_base_addr\modulenum -- numbytes )
```

The presence of both C and Forth prototypes conveys a lot of information. The C prototype explicitly states the type of each input and output parameter. The Forth prototype provides a descriptive name for the output parameter (a detail that is missing from the C prototype).

Let's look at the sample C prototype first. The leading **uint** tells us that a 16-bit unsigned integer is returned. The input parameter list starts with **xaddr xbuffer** which tells us that the first parameter is a 32-bit eXtended address (xaddress) as explained in the next section. Next, **addr mailbox_base_addr** tells us that the second parameter is a 16-bit address. The **int** modulenum parameter tells us that the final input is an integer specifying the Wildcard module number (corresponding to the hardware jumper address, as explained below).

The Forth prototype includes a standard "stack picture" in parentheses, with the input parameters to the left of the -- and the output parameter(s) to the right. A parameter with a leading 'x' is a 32-bit extended address, so we know that the first parameter in the example is a 32-bit xaddress that specifies a buffer. The '\' character in the stack picture is read as "under"; this character separates the stack items. The next input parameter is a 16-bit address in common memory. The final input parameter is the module number, specifying the Wildcard module number (corresponding to the hardware jumper address, as explained below). Each of the last 2 input parameters and the output parameter have the default 16-bit integer size: all Forth parameters are 16 bits unless otherwise indicated, as shown in the "Stack Symbols" table below.

Note that C is case-sensitive, while Forth is case-insensitive. For simplicity and clarity, both C and Forth versions of the functions in this glossary are spelled identically using the same case.

C Type Abbreviations Used in Function Declarations

Standard C type specifiers such as char, int, and long are used in the glossary declarations. In addition, we use four convenient typedefs that are defined in the **TYPES.h** header file:

```
typedef unsigned char     uchar;
typedef unsigned int      uint;
typedef unsigned long     ulong;
typedef unsigned long     xaddr;
```

The meanings of the first three typedefs are obvious; they are abbreviations for unsigned types. The **xaddr** typedef means "extended address", and is used when a 32-bit address parameter is passed. The least significant 16-bits of the xaddress specify the standard 16-bit machine address, and the most significant 16-bits specify the page.

Any other required C typedefs are described in the text of the relevant glossary entry.

Forth Stack Symbols

The following table describes the standard symbols used to represent items placed on the Forth data stack.

Stack Symbol	Size on Stack	Size in Memory	Valid Data Size	Meaning
addr	1 cell	2 bytes	2 bytes	16-bit address. Range: 0 to 65,535 (0-0xFFFF).

Stack Symbol	Size on Stack	Size in Memory	Valid Data Size	Meaning
page	1 cell	2 bytes	1 byte	Page. Range: 0 to 0x3F on PDQ platforms, 0 to 0xFF on earlier platforms.
xaddr	2 cells	4 bytes	3 bytes	32-bit extended address comprising an address and page: addr\page.
xmailbox	2 cells	4 bytes	3 bytes	Extended address of a mailbox variable.
xcfa	2 cells	4 bytes	3 bytes	Code field xaddr (execution address, corresponding to a 32-bit function pointer in C)
flag	1 cell	2 bytes	2 bytes	Boolean flag, 0 indicates false. Non-zero indicates true.
true	1 cell	2 bytes	2 bytes	Boolean flag, = -1 = 0xFFFF.
false	1 cell	2 bytes	2 bytes	Boolean flag, = 0
char	1 cell	1 byte	1 byte	ASCII character.
count	1 cell	2 bytes	2 bytes	A string count. While standard Forth strings are restricted to 1-byte counts, this library employs “long strings” and “long buffers” with 2-byte counts, supporting up to 65,533 bytes in the string.
cnt	1 cell	2 bytes	2 bytes	Synonym for count
bufsize	1 cell	2 bytes	2 bytes	A buffer size that constrains the maximum number of characters that can be stored in the buffer. 0-65,535.
xstring	2 cells	4 bytes	3 bytes	Extended address of the first data-containing byte of a string.
xbuffer	2 cells	4 bytes	3 bytes	Extended address of a buffer.
xbuffer	2 cells	4 bytes	3 bytes	Extended address of a buffer.
xlbuffer	2 cells	4 bytes	3 bytes	Extended address of a “long buffer” comprising a 2-byte count followed by the buffer data.
n	1 cell	2 bytes	2 bytes	Signed 16-bit (single) integer. Range: -32,768 to 32,767.
u	1 cell	2 bytes	2 bytes	Unsigned 16-bit (single) integer range: 0 to 65,535.
d	2 cells	4 bytes	4 bytes	32-bit integer. Used to represent encryption keys.
task_base	1 cell	2 bytes	2 bytes	16-bit address. Range: 0 to 65,535 (0-0xFFFF), the base address in common RAM of a task area (typically 1 Kbyte long).
timeout_msec	1 cell	2 bytes	2 bytes	Unsigned 16-bit (single) integer range: 0 to 65,535, representing the number of milliseconds before a timeout for the operation.
modulenum	1 cell	2 bytes	1 byte	Integer that specifies the module number (hardware address) of a Wildcard I/O module. Range: 0 to 7. The modulenum must correspond to the 2-bit hardware jumper setting of the target Wildcard combined with its Wildcard bus position (0 or 1) as the top bit of the 3-bit code.

Selecting the Module Address

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

The Module 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 modules. Module Port 0 provides access to modules 0-3 while Module Port 1 provides access to modules 4-7. Two modules on the same port cannot have the same address (jumper settings). Consult the EtherSmart/WiFi Wildcard User Guide for more information.

Ethernet and WiFi

The EtherSmart Wildcard uses an Ethernet hardware implementation called the "XPort" made by Lantronix. "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.

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 functions described in this document can operate on both the EtherSmart and WiFi Wildcards. Because the EtherSmart Wildcard was produced first and the names were left intact, the majority of these 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.

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. In fact, **WiFi_Setup** simply calls **Ether_Setup**, and then invokes **WiFi_Module** to declare the specified module as a WiFi Wildcard.

> ***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 the "Initialization" section in the categorized function list 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 words "Ethernet" and the prefix "Ether" as referring to the TCP/IP communications link which can be implemented via Ethernet or WiFi.

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 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 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 glossary entry for the **`WiFi_Security`** function 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.

Function Naming Conventions

Functions that provide basic Ethernet functionality, including configuration, 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`** 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. The latter functions include a call to the **`WiFi_Module`** function. See the "Initialization" section in the categorized function list 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.

Browser Notes

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, text/html and image/bmp) are present in a single webpage. 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.

We highly recommend the use of the free Opera web browser available for download at www.opera.com. Simply go to www.opera.com and select "Download Opera". The download and install are quick, and the program is compact. And, it's very easy to configure for the single-connection EtherSmart/WiFi Wildcard 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 the Opera web browser with the EtherSmart/WiFi Wildcard dynamic webserver.

Categorized List of EtherSmart/WiFi Library Functions

Configuration and Diagnostics (see also Initialization and WiFi Configuration)

ETHER_BUFSIZE_DEFAULT
Ether_DHCP_Name
Ether_Encryption
Ether_Gateway
Ether_Gateway_IP_Ptr
Ether_Info
Ether_Internal_Webserver_Port
Ether_IP_Info_Report
Ether_IP_Info_Request
Ether_Local_IP
Ether_Local_Port
ETHER_MIN_BUFFER_SIZE
Ether_My_IP_Ptr
Ether_Netmask_Ptr
Ether_Ping_Report
Ether_Ping_Request
Ether_Remote_IP_Ptr
Ether_Remote_Port_Ptr
Ether_Set_Inbuf
Ether_Set_Outbuf
Ether_Shutdown
Ether_TCP_Control
Ether_Telnet_Password
Ether_XPort_Defaults
Ether_XPort_Update
HTTP_AUTOSERVE_DEFAULT_ROWS
HTTP_INBUFSIZE_DEFAULT
HTTP_Is_Autoserve_Array
HTTP_OUTBUFSIZE_DEFAULT
HTTP_Set_Inbuf
HTTP_Set_Outbuf
HTTP_OUTBUFSIZE_DEFAULT

GUI Toolkit Functions for Remote Front Panel

(summarized in a separate glossary section at the end of this document)

Globalize_TVars
Graphic_To_Image
Has_Screen_Changed
Screen_Has_Changed
Screen_To_Image
Simulate_Touch
Simulated_Touch_To_Image

HTTP Webserver

Ether_Command_Manager
Ether_Connection_Manager
Ether_Error
Ether_Error_Clear
Ether_Service_Loop
ether_service_module
HTTP_Outbuf
HTTP_Outbuf_Cat
HTTP_Outbufsize
HTTP_OUTBUFSIZE_DEFAULT
HTTP_Parse_URL
HTTP_Plus_To_Space

HTTP Webserver

HTTP_Add_Handler
HTTP_AUTOSERVE_DEFAULT_ROWS
HTTP_Autoserve_Ptr
HTTP_BINARY_DATA_CONTENT
HTTP_Default_Handler
HTTP_Default_Handler_Ptr
HTTP_Enable_Ptr
HTTP_Fieldname_Count
HTTP_Fieldname_Ptr
HTTP_Get_Timeout_Msec_Ptr
HTTP_IMAGE_BITMAP_CONTENT
HTTP_IMAGE_GIF_CONTENT
HTTP_IMAGE_JPEG_CONTENT
HTTP_IMAGE_PNG_CONTENT
HTTP_Inbuf
HTTP_Inbufsize
HTTP_INBUFSIZE_DEFAULT
HTTP_Index_Ptr
HTTP_Is_Autoserve_Array
HTTP_Numbytes_Sent
HTTP_Put_Content_Type
HTTP_Put_Header
HTTP_Send_2Buffers
HTTP_Send_Buffer
HTTP_Send_LBuffer
HTTP_Server
HTTP_Set_Inbuf
HTTP_Set_Outbuf
HTTP_Status_Ptr
HTTP_TEXT_HTML_CONTENT
HTTP_TEXT_PLAIN_CONTENT
HTTP_Timeout_Msec_Ptr
HTTP_To_Next_Field
HTTP_Unescape
HTTP_URL_Base_Count
HTTP_URL_Full_Count
HTTP_URL_Ptr
HTTP_Value_Count
HTTP_Value_Ptr

HTTP/GUI Webserver for Remote Front Panel (see also HTTP Webserver)

Ether_Check_GUI
HTTP_Add_GUI_Handler
HTTP_GUI_Send_2Buffers
HTTP_GUI_Send_Buffer
HTTP_GUI_Send_LBuffer
HTTP_Imagemap

Initialization (see also Configuration and Diagnostics)

Ether_Info_Init
Ether_Init
Ether_Setup
Ether_Setup_Default
Ether_Task_Setup
WiFi_Check
WiFi_Info_Init
WiFi_Init
WiFi_Module
WiFi_Setup
WiFi_Setup_Default
WiFi_Task_Setup

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.

Mailboxes

ether_command
ether_gui_message
ether_response

Revectored Serial via Ethernet

E_ASCII_Key
E_Ask_Key
E_Emit
E_Key
Ether_ASCII_Key
Ether_Ask_Emit
Ether_Ask_Key
Ether_Emit
Ether_Key
Ether_Monitor
ether_revector_module
Ether_Serial_Revector
XTERM/38400SSP (Forth only)

Serial Tunneling and Email

Ether_Add_Chars
Ether_Add_Data
Ether_Add_Line
Ether_Await_Response
ETHER_BUFSIZE_DEFAULT
Ether_Check_Response
Ether_Command_Manager
Ether_Connect
Ether_Connect_Status
Ether_Connection_Manager
Ether_Disconnect
Ether_Disconnect_During_Send
Ether_DISCONNECT_Flush
Ether_Error
Ether_Error_Clear
ETHER_Flush
Ether_Flush_NBytes
Ether_Get_Chars
Ether_Get_Data
Ether_Get_Line
Ether_Inbuf
Ether_Inbufsize
ETHER_MIN_BUFFER_SIZE
Ether_Outbuf
Ether_Outbuf_Cat
Ether_Outbufsize
Ether_Passive_Non_Web_Connection
Ether_Ready_For_Command
Ether_Send_2Buffers
Ether_Send_Buffer
Ether_Send_Email
Ether_Send_LBuffer
Ether_Service_Loop
ether_service_module
Ether_Set_Inbuf
Ether_Set_Outbuf
Ether_Tunnel_Enable_Ptr

String Primitives

Cat
LCOUNT (Forth only)

String Primitives

Ether_Outbuf_Cat
HTTP_Outbuf_Cat
LPARSE (Forth only)

WiFi Configuration (see also Initialization)

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

EtherSmart/WiFi Wildcard Function Glossary

> ***Note***
>
> All functions can operate on both the EtherSmart and WiFi Wildcards, unless:
>
> 1. the glossary entry says otherwise, or,
> 2. the function starts with the "**WiFi_**" prefix.
>
> The WiFi Wildcard driver is an extension of the EtherSmart Wildcard driver, and the original names were left intact. Thus the "Ether" and "Ethernet" function names should be interpreted as referring to the TCP/IP communications link which can be implemented via Ethernet or WiFi.

C: void **Cat** (xaddr xcountedLString, uint umaxChars, xaddr xstringToAdd, int countToAdd, uint eol)
4th:**Cat** (xcountedLString\umaxChars\xstringToAdd\countToAdd\eol --)

Concatenates (appends) a string specified by xstringToAdd and countToAdd, plus a specified 1- or 2-byte eol (end of line) sequence and a terminating null byte (not included in the count), to a string in a long buffer specified by xcountedLString, and increments the buffer count that is stored in the first two bytes of xcountedLString to reflect the updated buffer contents. This routine clamps the maximum number of bytes in the destination buffer at xcountedLString to umaxChars; this value does not include the 2-byte count stored at the start of the buffer. (Recall that a longstring buffer comprises a 2-byte count followed by the buffer contents). The allocated buffer size in RAM should be at least 3 bytes larger than the specified umaxChars to allow for the 2-byte count and the terminating null byte. While umaxChars can be as large as 65532 bytes, the countToAdd string length must be a positive integer less than or equal to 32,767 bytes. The eol parameter can contain 1 or 2 bytes; typical values are 0x0D (carriage return), 0x0A (linefeed), 0x0D0A (carriage return/linefeed), or 0x00 (null byte). If the eol parameter = -1, no eol bytes are stored in the string by this function. If the most significant (ms) byte of the eol parameter is non-zero, two bytes are stored in the buffer after the appended string: first the msbyte of eol, then the lsbyte of eol. This function provides a way for both forth and C programs to build a long string in memory.

NOTE: Remember to zero the first 2 bytes (16-bit count) of the xcountedLString buffer before the first call to this routine; this initializes the counted lbuffer to its starting size of zero.

C: uchar **E_ASCII_Key** (void)
4th: **E_ASCII_Key** (-- char)

For the EtherSmart/WiFi module whose value is stored in the ether_revector_module variable, waits for and returns the next pending input character. Unlike E_Key, this function ignores any incoming character that has its most significant bit set, or that is non-printable. It ignores "control" characters with values less than 0x20, except that ascii 0x07 through 0x0A (BEL, BS, TAB, LF) or 0x0D (CR) are accepted. This function is useful for ascii-only file transfers, but of course must not be used for binary transfers. See Ether_Serial_Revector and Ether_Monitor.

Implementation detail: Fetches the contents of ether_revector_module and calls Ether_ASCII_Key.

C: int **E_Ask_Key** (void)
4th: **E_Ask_Key** (-- flag)
For the EtherSmart/WiFi module whose value is stored in the ether_revector_module variable, returns a flag that is true if the 64-byte UART input FIFO (First In/First Out buffer) on the Wildcard contains at least one character, and is false there are no characters in the input FIFO. See Ether_Serial_Revector and Ether_Monitor.
Implementation detail: Fetches the contents of ether_revector_module and calls Ether_Ask_Key.

C: void **E_Emit** (uchar character)
4th: **E_Emit** (character --)
Sends the specified character for transmission via TCP/IP to the EtherSmart/WiFi Wildcard module whose value is stored in the ether_revector_module variable. This function waits until the UART on the Wildcard can accept a character, then sends it. This function has the same stack picture (parameter list) as the standard Emit output function used by the operating system, and so can be used to revector serial I/O via the EtherSmart/WiFi Wildcard. See Ether_Serial_Revector and Ether_Monitor.
Implementation detail: Fetches the contents of ether_revector_module and calls Ether_Emit.

C: uchar **E_Key** (void)
4th: **E_Key** (-- flag)
For the EtherSmart/WiFi module whose value is stored in the ether_revector_module variable, waits for and returns the next pending input character. See Ether_Serial_Revector and Ether_Monitor.
Implementation detail: Fetches the contents of ether_revector_module and calls Ether_Key.

C: void **Ether_Add_Char**s (xaddr xlbuffer, uint maxbytes, uint maxlines, char eol,
int discard_alt_eol, int discard_msbit_set, uint timeout_msec, int modulenum)
4th: **Ether_Add_Chars** (xlbuf\maxchars\maxlines\eol\discard.alt.eol\no.msbitset\timeout\module--)
Stores the specified input parameters into the ether_info struct and SENDs a message via the ether_command mailbox to the task running the Ether_Service_Loop which dispatches the action function. (If the mailbox is full because the action task hasn't cleared it yet, the SEND routine will PAUSE; see Ether_Ready_For_Command). 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 xlbuf+2+prior_count, where prior_count is the 16-bit contents at xlbuf 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_char = '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 tty 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. The specified maxbytes must be less than or equal to 65533 so that the maximum buffersize including count fits in a 16-bit number. After calling this routine the application must clear the ether_response mailbox using Ether_Check_Response or Ether_Await_Response. This routine must not be used to input binary (non-ascii) data; see also Ether_Get_Data and Ether_Add_Data.

C: void **Ether_Add_Data** (xaddr xlbuffer, uint maxbytes, uint timeout_msec, int modulenum)

4th: **Ether_Add_Data** (xlbuffer\maxbytes\timeout_msec\modulenum --)

Stores the specified input parameters into the ether_info struct and SENDs a message via the ether_command mailbox to the task running the Ether_Service_Loop which dispatches the action function. (If the mailbox is full because the action task hasn't cleared it yet, the SEND routine will PAUSE; see Ether_Ready_For_Command). Requests and stores incoming 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 xlbuf+2+prior_count, where prior_count is the 16-bit contents at xlbuf 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. This function exits within the specified timeout_msec whether or not the maximum number of bytes 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. The specified maxbytes must be less than or equal to 65533 so that the maximum buffersize including count fits in a 16-bit number. After calling this routine the application must clear the ether_response mailbox using Ether_Check_Response or Ether_Await_Response. See also Ether_Get_Data and Ether_Add_Chars.

C: void **Ether_Add_Line** (xaddr xlbuffer, uint maxbytes, char eol, int discard_alt_eol,
int discard_msbit_set, uint timeout_msec, int modulenum)

4th: **Ether_Add_Line** (xlbuffer\maxchars\eol\discard.alt.eol?\no.msbitset?\timeout_msec\module--)

Calls Ether_Add_Chars, passing 1 as the maxlines parameter to input a maximum of one line of text with the specified eol (end of line) character. See Ether_Add_Chars.

C: uchar **Ether_ASCII_Key** (int modulenum)
4th: **Ether_ASCII_Key** (modulenum -- char)
For the specified EtherSmart/WiFi module, waits for and returns the next pending input character. Unlike Ether_Key, this function ignores any incoming character that has its most significant bit set, or that is non-printable. It ignores "control" characters with values less than 0x20, except that ascii 0x07 through 0x0A (BEL, BS, TAB, LF) or 0x0D (CR) are accepted. This function is useful for ascii-only file transfers, but of course must not be used for binary transfers. See Ether_ Key, Ether_Emit, and E_Ascii_Key.

C: int **Ether_Ask_Emit** (int modulenum)
4th: **Ether_Ask_Emit** (modulenum -- ok_to_send?)
For the specified EtherSmart/WiFi module, returns a flag that is true if the 64-byte UART output FIFO (First In/First Out buffer) on the Wildcard is ready to accept a character, and is false otherwise. See Ether_Emit, Ether_Ask_Key, and E_Ask_Emit.

C: int **Ether_Ask_Key** (int modulenum)
4th: **Ether_Ask_Key** (modulenum -- char_available?)
For the specified EtherSmart/WiFi module, returns a flag that is true if the 64-byte UART input FIFO (First In/First Out buffer) on the Wildcard contains at least one character, and is false there are no characters in the input FIFO. See Ether_Emit, Ether_Key, and E_Ask_Key.

C: ulong **Ether_Await_Response** (int modulenum)
4th: **Ether_Await_Response** (modulenum -- d.mailbox_contents)
This blocking function waits until the ether_response mailbox contains a non-zero value, and returns the value. Use this function to wait until a just-issued command to the Ethernet task has completed and generated a response in the ether_response mailbox for the specified EtherSmart/WiFi modulenum. The ether_response mailbox is declared using the Ether_Info_Init function and its callers Ether_Init, Ether_Setup, Ether_Setup_Default, and Ether_Task_Setup. For a WiFi Wildcard the ether_response mailbox is declared using the WiFi_Info_Init function and its callers WiFi_Init, WiFi_Setup, WiFi_Setup_Default, and WiFi_Task_Setup. Typically the least significant 16-bits in the mailbox is either an error code (also stored at Ether_Error) or a number of bytes sent or received. The most significant byte in the mailbox is the command_id, and the remaining byte is the modulenum. For the recommended non-blocking version and additional details, see Ether_Check_Response.

C: **ETHER_BUFSIZE_DEFAULT**
4th: **ETHER_BUFSIZE_DEFAULT**
A 16-bit constant that returns the value 510. This constant is used by Ether_Setup and WiFi_Setup to specify the size of the Ether_Outbuf and Ether_Inbuf buffers, and is returned by Ether_Inbufsize and Ether_Outbufsize after Ether_Setup or WiFi_Setup is executed. Each buffer is allocated as 512 bytes, comprising a 2-byte count stored in the first 2 bytes of the buffer, followed by the 510 maximum bytes of data in the buffer.

C: int **Ether_Check_GUI** (int modulenum)
4th: **Ether_Check_GUI** (modulenum -- gui_handler_called?)
This non-blocking function checks the contents of the ether_gui_message mailbox without pausing. If an application supports a web-based "remote front panel" for an instrument that contains a touchscreen and graphics display, this function must be called on each pass through the application task's program loop. If the ether_gui_message mailbox is empty (i.e., if its

contents = zero indicating that there is no pending GUI web request), then this routine returns a 16-bit zero flag. If the mailbox is not empty (i.e., if its contents = the xcfa and modulenum packed into a 32-bit parameter indicating that there is a pending GUI web request), then this routine executes the specified GUI handler and returns a 16-bit true (-1) flag to indicate that the handler was called. In this case, the Ethernet task automatically closes the HTTP connection after the handler has executed. No additional mail to the application task is generated by the execution of the handler function. The handler function for the web request is posted using the HTTP_Add_GUI_Handler function; see its glossary entry for important details about the handler function and URL web address. A typical application program that supports a web-based "remote front panel" will call this routine and Ether_Check_Response and Ether_Connect_Status on each pass through the application program loop to manage the Ethernet activities.

Usage Notes: A GUI handler is used to implement a web-based "remote front panel" for an instrument that contains a touchscreen and graphics display. When a user clicks on a screen image (typically the same image that is present on the graphics display) presented in a browser window, a series of events is launched that result in the GUI toolkit processing the input as if a touch on the touchscreen had occurred, calling the associated GUI action function, and updating the screen image both on the instrument an on the remote browser screen. A GUI handler is different than a standard webservice handler. A standard non-GUI webservice handler executes automatically without any intervention from the user's application task, and is dispatched by the Ethernet task running the Ether_Service_Loop routine. A GUI handler, on the other hand, requires an interaction with the GUI Toolkit, and, for task synchronization reasons, must be dispatched from the user's application task. The HTTP_Add_GUI_Handler function posts the handler function and the associated URL to the autoserve array, and marks the function as a GUI Handler so that it will be dispatched via the application task from this Ether_Check_GUI routine. In the HTTP GUI handler function, the HTTP_Imagemap function (see its glossary entry) should be called to extract and return the X and Y coordinates from the query field. Then the Simulated_Touch_To_Image function (defined in the special GUI Toolkit section of this glossary document) can be invoked to simulate the touch at the specified screen coordinates, draw the press and release graphics, activate the screen button's press handler, and, if the screen image changed, reload the screen image buffer with the updated screen bitmap image. Then HTTP_GUI_Send_2Buffers can be used to send the HTTP header and HTML text (in the first buffer) and graphics image (in the second buffer) to the browser to complete the handler's actions.

C: ulong **Ether_Check_Response** (int modulenum)

4th: **Ether_Check_Response** (modulenum -- d.mailbox_contents)

This non-blocking function checks the contents of the ether_response mailbox for the specified module without waiting or pausing. The user's main application program can call this routine repeatedly as part of the main program loop to keep the response mailbox clear. If mail is pending from the Ethernet control task running the Ether_Service_Loop, this routine returns the mailbox contents; this occurs when the most recently-issued command has generated a response in the ether_response mailbox. If the mailbox is empty, the returned value is a 32-bit zero. Use this function to periodically check the response in the ether_response mailbox whose address is stored in the ether_info struct of the specified wildcard module. The location of the ether_response mailbox is set by the EtherSmart initialization functions Ether_Info_Init, Ether_Init, Ether_Setup, Ether_Setup_Default, and Ether_Task_Setup; or by the WiFi Wildcard initialization functions WiFi_Init, WiFi_Setup, WiFi_Setup_Default, and WiFi_Task_Setup. Typically the least significant 16-bits in the mailbox is either an error code (also stored at

Ether_Error) or a number of bytes sent or received. The most significant byte in the mailbox is the command_id, and the remaining byte is the modulenum. A typical application program calls this routine, and Ether_Connect_Status and/or Ether_Passive_Non_Web_Connection on each pass through the program loop to manage the Ethernet activities performed by the task running the Ether_Service_Loop. Applications that support the web-based "remote front panel" would also call Ether_Check_GUI. Note that the ether_response mailbox is not actively "cleared" (emptied) unless this routine returns a non-zero result, indicating that an incoming message has been received. For a blocking version, see Ether_Await_Response.

C: ulong **ether_command**
4th: **ether_ command** (-- xaddr)

A mailbox in common RAM that conveys a 32-bit value from the user's application task to the Ethernet control task. The programmer never has to explicitly access this mailbox, as it is zeroed by the Ether_Init (or WiFi_Init) routines and their calling initialization functions; the mailbox is managed by the Ethernet driver functions. Each command function writes a command_id byte and the relevant modulenum into the msword, and 0 into the lsword of this mailbox. The Ether_Command_Manager routine called by Ether_Service_Loop receives the mailbox contents and dispatches the correct action function from the Ethernet task.

Note: This mailbox is typically not accessed directly. In C, this mailbox can be used as an lvalue or an rvalue, just like any C variable. In Forth, this variable has the standard behavior or returning the xaddress of its contents.

C: void **Ether_Command_Manage**r (int modulenum)
4th: **Ether_Command_Manager** (modulenum --)

This is the command processor, called by Ether_Service_Loop which is the task activation routine for the Ethernet control task. This routine is typically not called by the programmer. Does nothing if the ether_command mailbox is empty. If the ether_command mailbox is full and it contains the specified modulenum, this routine calls the routine corresponding to the command_id parameter, with the called function expecting its required parameters to be in the ether_info struct, as placed by the originating function invoked from the application program. Commands dispatched by this function include those originated by the following functions:

Ether_Get_Data	Ether_Add_Data
Ether_Get_Chars	Ether_Add_Chars
Ether_Send_Buffer	Ether_Send_LBuffer
Ether_Send_2Buffers	HTTP_GUI_Send_Buffer
HTTP_GUI_Send_LBuffer	HTTP_GUI_Send_2Buffers
Ether_Connect	Ether_Disconnect
Ether_Flush	Ether_Flush_Nbytes
Ether_Ping_Request	Ether_Ping_Report
Ether_IP_Info_Request	Ether_IP_Info_Report
Ether_XPort_Defaults	Ether_XPort_Update
Ether_Send_Email	

See their glossary entries for details.

C: void **Ether_Connect** (char ip1, char ip2, char ip3, char ip4, int port,
int timeout_msec, int modulenum)
4th: **Ether_Connect** (ip1\ip2\ip3\ip4\port\timeout_msec\module --)

Stores the specified input parameters into the ether_info struct and SENDs a message via the ether_command mailbox to the task running the Ether_Service_Loop which dispatches the

action function. (If the mailbox is full because the action task hasn't cleared it yet, the SEND routine will PAUSE; see Ether_Ready_For_Command). This function opens a TCP/IP connection to the specified remote port at remote IP address ip1.ip2.ip3.ip4. If the connection is not established within the specified timout_msec, the connection attempt is abandoned and a non-zero ERROR_INVALID_RESPONSE = 0x08 (see Ether_Error) result is returned in the ether_response mailbox and in Ether_Error. If there was already a connection established when this routine was called, the ERROR_ALREADY_CONNECTED = 0x20 (see Ether_Error) result is returned in the ether_response mailbox and in Ether_Error. If we were not connected upon entry into this routine, Ether_Connect_Status (see its glossary entry) is updated. 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 error flag in the remaining 2 bytes is placed in the ether_response mailbox; the error flag is also available using Ether_Error. The error flag is zero if the operation was successful; otherwise the error is nonzero. After calling this routine the application must clear the ether_response mailbox using Ether_Check_Response or Ether_Await_Response.

C: int **Ether_Connect_Status** (int modulenum)

4th: **Ether_Connect_Status** (modulenum -- status)

Returns an integer representing the current connection status of the specified EtherSmart Wildcard as managed by the task running the Ether_Service_Loop. The returned values have the following meanings:

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

Web connections are typically transient, as they are closed by the webserver running on the Ethernet task after the user-specified handler corresponding to the requested URL has executed. Active connections (those initiated by the application program) must be maintained by the application program by calling routines to accept incoming data, send outgoing data, and close the connection at the appropriate time. Calling this Ether_Connect_Status routine enables the application task to detect incoming passive non-web connections from the remote so that the connection can be serviced by calling routines to accept incoming data, send outgoing data, and close the connection at the appropriate time. Even while an active or passive non-HTTP connection is open, this routine should be periodically called to detect if the remote has unexpectedly closed the connection. For a function that directly reports whether a passive non-web connection is active, see Ether_Passive_Non_Web_Connection. If Ether_Connect_Status indicates that a formerly open connection has closed, (for example, if the remote terminated the connection), it is recommended that you input any incoming chars using Ether_Get_Data or a related function, or call Ether_Flush to be sure you've cleaned up any incoming bytes so that they do not erroneously appear as the first bytes sent by the subsequent connection. If the returned status is one of the odd values 3, 5, or 7, it indicates that the remote either issued a RST (reset) packet, or simply closed the connection while we were attempting to send data to the remote. If the remote closed the connection, the connection status will automatically change

to 0 (not connected). If, however, the remote reset the connection and immediately reconnected, further sends via Ether_Send_Buffer, Ether_Send_LBuffer, and Ether_Send_2Buffers will be ineffective until the connection status changes (for example, via the execution of Ether_Disconnect followed by Ether_Connect). This behavior prevents inadvertently sending data meant for one TCP/IP connection to another one.

C: void **Ether_Connection_Manager** (int modulenum)
4th: **Ether_Connection_Manager** (modulenum --)
This is the connection status manager, called by Ether_Service_Loop (see its glossary entry) which is the task activation routine for the Ethernet control task (see Ether_Task_Setup and WiFi_Task_Setup). This routine is typically not called by the programmer. Checks and updates the connection status and corresponding flags in the ether_info struct, and detects and identifies passive incoming connections as HTTP- or non-HTTP connections. If the incoming connection is identified as an HTTP connection by virtue of a leading GET substring, and if the variable pointed to by HTTP_Enable_Ptr is non-zero, this routine calls HTTP_Server (which calls the user-specified handler function to serve out the web content) and closes the connection. If the connection identifies as non-HTTP, and if the variable pointed to by Ether_Tunnel_Enable_Ptr is non-zero, the connection status is set to PASSIVE_NON_WEB_CONNECTION, and it is up to the user's application task repeatedly calling Ether_Connect_Status and/or Ether_Passive_Non_Web_Connection on each pass through the program loop to detect and service the incoming serial tunneling connection. The variables pointed to by HTTP_Enable_Ptr and Ether_Tunnel_Enable_Ptr are set to -1 (true) by Ether_Info_Init and WiFi_Info_Init and their higher level calling functions as listed in the "Initialization" section of the categorized function list.

C: void **Ether_DHCP_Name** (xaddr xname_string, int name_cnt, int modulenum)
4th: **Ether_DHCP_Name** (xname_string\count\modulenum --)
Writes the specified string specifier parameters into the ether_info struct for the specified module as the name used by DHCP (Dynamic Host Configuration Protocol). The xaddress xname_string is the 32-bit base address of the first character of the string, and count is the number of bytes in the string (clamped to a maximum of 8 bytes). Assuming that you have initialized the Ethernet task (see Ether_Task_Setup and WiFi_Task_Setup), you can instantiate the string into Lantronix flash after invoking this function by executing Ether_XPort_Update followed by Ether_Await_Response. Use of this string is optional. If you do not change the DHCP name from its default (as set by Ether_XPort_Defaults) and the local IP is stored as its default value of 0.0.0.0, then the DHCP name defaults to Cxxxxxx, where xxxxx represents the last 6 digits of the MAC address shown on the label on the Lantronix XPort or WiPort. For example, if the MAC address is 00-20-4A-12-34-56, then the default DHCP name is C123456. Note that the DHCP name is not a nameservice name, and cannot in general be typed into a web browser's address bar in place of the IP address. See the user guide for hints about how to refer to the EtherSmart/WiFi Wildcard by name on the Local Area Network.

C: void **Ether_Disconnect** (int modulenum)
4th: **Ether_Disconnect** (modulenum --)
SENDs a message via the ether_command mailbox to the task running the Ether_Service_Loop which dispatches the action function. (If the mailbox is full because the action task hasn't cleared it yet, the SEND routine will PAUSE; see Ether_Ready_For_Command). This function terminates the current connection. If there is a TCP/IP connection on entry into this routine and we successfully disconnect, or if there is no active TCP/IP connection on entry into this routine, Ether_Connect_Status (see its glossary entry) is updated to return the disconnected state (0).

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 error flag in the remaining 2 bytes is placed in the ether_response mailbox; the error flag is also available using Ether_Error. The error flag is zero if the operation was successful, or nonzero if there is still a TCP/IP connection after attempting to hang-up; this can happen when we've sent a lot of chars to the Lantronix hardware, as the Lantronix will have to send them all before disconnecting. Therefore, a non-zero error flag does not necessarily indicate that there is a problem, but if you want to ensure a disconnect, you can call this function again if a nonzero error flag is returned. After calling this routine the application must clear the ether_response mailbox using Ether_Check_Response or Ether_Await_Response, but note that on the EtherSmart Wildcard the result may not be present until over 2 seconds have elapsed, so please be patient. This routine does not flush the input FIFO (First In/First Out) buffers; see Ether_Disconnect_Flush for a routine that does flush the input buffers.

Implementation detail: On the WiFi Wildcard, there is a hardware line that, when strobed by the Mosaic controller, terminates the TCP/IP connection. On the EtherSmart Wildcard, there is no such hardware line, so a slower disconnect approach is used. The EtherSmart Wildcard disconnect sequence requires this function to wait 1 second, send to the XPort the 3-byte escape sequence preamble +++, then wait another second, then send the ATH hang-up sequence followed by a carriage return. If the remote computer disconnects or sends a RST (TCP/IP reset) packet during the disconnect process, this function aborts the sending of the escape sequence; this is dictated by hardware and firmware constraints in the Lantronix XPort.

C: int **Ether_Disconnect_During_Send** (int modulenum)

4th: **Ether_Disconnect_During_Send** (modulenum -- flag)

Returns a true flag if a disconnect was detected while we were attempting to send data via TCP/IP. This routine is called within the data send routines associated with Ether_Send_Buffer, Ether_Send_LBbuffer, Ether_Send_2Buffers, HTTP_Send_Buffer, HTTP_Send_LBuffer, HTTP_Send_2Buffers, HTTP_GUI_Send_Buffer, HTTP_GUI_Send_LBuffer, and HTTP_GUI_Send_2Buffers. If a transient disconnect occurred (e.g., owing to a RST reset packet from the remote), further data transmission will be suppressed until the connection status changes (for example, via the execution of Ether_Disconnect followed by Ether_Connect). This behavior prevents inadvertently sending data meant for one TCP/IP connection to another one. See Ether_Connect_Status.

Implementation detail: This routine's out is a result of monitoring of the delta_dcd hardware bit on the UART chip of the EtherSmart/WiFi Wildcard.

C: void **Ether_Disconnect_Flush** (int modulenum)

4th: **Ether_Disconnect_Flush** (modulenum --)

SENDs a message via the ether_command mailbox to the task running the Ether_Service_Loop which dispatches the action function. (If the mailbox is full because the action task hasn't cleared it yet, the SEND routine will PAUSE; see Ether_Ready_For_Command). This function terminates the current connection. It also reads and discards (flushes) any incoming characters from the input FIFO (First In/First Out) buffers on the EtherSmart/WiFi Wildcard, so be sure you've already retrieved any expected incoming bytes. If there is a TCP/IP connection on entry into this routine and we successfully disconnect, or if there is no active TCP/IP connection on entry into this routine, Ether_Connect_Status (see its glossary entry) is updated to return the disconnected state (0). 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 error flag in the remaining 2 bytes is placed in the ether_response

mailbox; the error flag is also available using Ether_Error. The error flag is zero if the operation was successful, or nonzero if there is still a TCP/IP connection after attempting to hang-up; this can happen when we've sent a lot of chars to the Lantronix hardware, as the Lantronix will have to send them all before disconnecting. Therefore, a non-zero error flag does not necessarily indicate that there is a problem, but if you want to ensure a disconnect, you can call this function again if a nonzero error flag is returned. After calling this routine the application must clear the ether_response mailbox using Ether_Check_Response or Ether_Await_Response, but note that on the EtherSmart Wildcard the result may not be present until over 2 seconds have elapsed, so please be patient. See Ether_Disconnect for a routine does not flush the input FIFO buffers.

Implementation detail: On the WiFi Wildcard, there is a hardware line that, when strobed by the Mosaic controller, terminates the TCP/IP connection. On the EtherSmart Wildcard, there is no such hardware line, so a slower disconnect approach is used. The EtherSmart Wildcard disconnect sequence requires this function to wait 1 second, send to the XPort the 3-byte escape sequence preamble +++, then wait another second, then send the ATH hang-up sequence followed by a carriage return. If the remote computer disconnects or sends a RST (TCP/IP reset) packet during the disconnect process, this function aborts the sending of the escape sequence; this is dictated by hardware and firmware constraints in the Lantronix XPort.

C: void **Ether_Emit** (uchar character, int modulenum)
4th: **Ether_Emit** (character\modulenum --)

Sends the specified character to the specified EtherSmart/WiFi Wildcard module for transmission to the Ethernet. This function waits until the UART on the Wildcard can accept a character, then sends it. See Ether_Ask_Emit, Ether_Key, and E_Emit.

C: int **Ether_Encryption** (xaddr key_buffer_xbase, int num_key_bytes, int modulenum)
4th: **Ether_Encryption** (key_buffer_xbase\num_key_bytes\modulenum--error)

For the specified EtherSmart/Wifi module, installs 32-byte (256-bit) AES Rijndael encryption with the encryption key stored at key_buffer_xbase. The lesser of num_key_bytes or 32 bytes are stored as the key, with any unspecified bytes set to zero so that the total key length (including any trailing zeros) is 32 bytes. The most significant byte of the key is the byte stored at key_buffer_xbase. Returns 0 if successful, or returns a nonzero flag if the operation was not successful. To undo the effect of this command and return to non-encrypted operation of the EtherSmart/WiFi Wildcard, make sure that the Ethernet task is running (see Ether_Task_Setup or WiFi_Task_Setup) and execute Ether_XPort_Defaults (see its glossary entry; it works for both EtherSmart and WiFi Wildcards). Rijndaell 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, while preventing anyone who does not know the encryption key from deciphering the network data.

To use: After a restart, initialize the EtherSmart/WiFi Wildcard using Ether_Setup or WiFi_Setup one of their calling functions listed in the "Initialization" section of the categorized function list, and then invoke Ether_Encryption, passing it the base xaddress and size of the key which is 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.

Note: This AES encryption is independent of and not related to the WiFi encryption on the WiFi Wildcard. See the WiFi_Security and WiFi_Encryption glossary entries for more details.

Implementation detail: This routine acts directly on the Lantronix hardware, not via the ether task, and so should be executed when there is no network activity being managed by the

Ethernet task. This function suspends multitasking for several seconds to enter the monitor mode, resumes multitasking, moves the default block1 contents to Ether_Outbuf, writes the encryption keys and security flags to enable 256-bit AES encryption, writes block1 to the Lantronix device, then executes the monitor mode RS reset command to instantiate the new values into the Lantronix flash memory. The entire operation takes approximately 13 seconds, so please be patient.

Encryption notes: An alternate way to configure the encryption settings is to go into setup mode by connecting to port 9999 using the "raw" data transfer mode of a free TCP/IP terminal program such as Putty, and hitting enter within 3 seconds. Then choose option 6 (security), then follow the prompts to enable encryption, choose key length, and change/enter a key. Providing the same key to several Lantronix devices on a network enables them to exchange encrypted communications among themselves.

Export agreement: 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.

C: uint **Ether_Error** (int modulenum)
4th: **Ether_Error** (modulenum -- u)

Returns the contents of the 16-bit error flag located in the ether_info structure for the specified EtherSmart/WiFi module. This flag is set by a number of the functions in this glossary at the same time the ether_response mailbox is written to, and in some cases the error code is also written as the least significant 16-bits in ether_response. Ether_Error is cleared by Ether_Error_Clear. The numeric (unnamed) error codes are as follows:

Value	Meaning	
0x08	ERROR_INVALID_RESPONSE	// expected prompt or string not received
0x10	ERROR_NO_RESPONSE	// no response from remote
0x20	ERROR_ALREADY_CONNECTED	// only 1 connection is allowed at a time
0x40	ERROR_CANT_DISCONNECT	// can't disconnect (for some reason)
0x80	ERROR_BUFFER_TOO_SMALL	// buffer is too small
0x100	ERROR_TIMEOUT	// timed out before operation could complete
0x200	ERROR_INVALID_PARAM	// out of range or invalid parameter

C: void **Ether_Error_Clear** (int modulenum)
4th: **Ether_Error_Clear** (modulenum --)

Writes 0 to the 16-bit Ether_Error error flag located in the ether_info structure for the specified EtherSmart/WiFi module. Caution: The Ether_Error flag is typically written to by the Ethernet task, so executing Ether_Error_Clear from a routine running in the application task area will be asynchronous with respect to the setting of the flag. See Ether_Error for a list of error codes.

C: void **Ether_Flush** (int modulenum)
4th: **Ether_Flush** (modulenum --)

Stores the specified input parameters into the ether_info struct and SENDs a message via the ether_command mailbox to the task running the Ether_Service_Loop which dispatches the action function. (If the mailbox is full because the action task hasn't cleared it yet, the SEND routine will PAUSE; see Ether_Ready_For_Command). Inputs and discards all incoming bytes present from the specified EtherSmart/WiFi Wildcard at the time this routine is invoked. As long as incoming characters are available, they are discarded. This routine 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 is to prevent an indefinite "hung" state). For a more limited flush of a specified number of bytes, see Ether_Flush_Nbytes. 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 0 in the remaining 2 bytes is placed in the ether_response mailbox. After calling this routine the application must clear the ether_response mailbox using Ether_Check_Response or Ether_Await_Response.

C: void **Ether_Flush_NBytes** (uint maxbytes, uint timeout_msec, int modulenum)
4th: **Ether_Flush_NBytes** (maxbytes\timeout_msec\modulenum --)
Stores the specified input parameters into the ether_info struct and SENDs a message via the ether_command mailbox to the task running the Ether_Service_Loop which dispatches the action function. (If the mailbox is full because the action task hasn't cleared it yet, the SEND routine will PAUSE; see Ether_Ready_For_Command). This function inputs and discards up to maxbytes incoming bytes from the specified EtherSmart/WiFi Wildcard. Exits within the specified timeout_msec whether or not the maximum number of bytes have been flushed. Use this routine when you know exactly how many bytes to flush (as with a command echo). 500 milliseconds is suggested as a reasonable default timeout value. For a more general and thorough flush see Ether_Flush. 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 0 in the remaining 2 bytes is placed in the ether_response mailbox. After calling this routine the application must clear the ether_response mailbox using Ether_Check_Response or Ether_Await_Response.

C: void **Ether_Gateway** (char ip1, char ip2, char ip3, char ip4, int subnet_bits, int modulenum)
4th: **Ether_Gateway** (ip1\ ip2\ ip3\ ip4\subnet_bits\modulenum --)
Sets the 32-bit gateway IP (Internet Protocol) address pointed to by Ether_Gateway_IP_Ptr in the ether_info struct to the specified 4-byte value ip1.ip2.ip3.ip4 for the specified EtherSmart/WiFi modulenum. Also sets the number of subnet bits for the Local Area Network (LAN) to the specified value. The gateway is the router computer on the LAN. The number of subnet bits is equal to the number of bits specified as zeros in the LAN's netmask. For example, to set the gateway IP address to 10.0.1.2 with 8 subnet bits, pass to this routine the IP parameters 10, 0, 1, and 2 followed by 8, followed by the EtherSmart/WiFi Wildcard modulenum. 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. Assuming that you have initialized the Ethernet task (see Ether_Task_Setup or WiFi_Task_Setup), you can implement the specified information so that it is stored in the Lantronix flash memory and used by the Lantronix device by executing this function and Ether_Local_IP (see its glossary entry). Then, to instantiate the values in Lantronix flash, execute Ether_XPort_Update followed by Ether_Await_Response. To use a local IP, gateway IP and netmask that are automatically assigned, specify these as 0.0.0.0 (the factory default); then the Lantronix device relies on DHCP (Dynamic Host Configuration Protocol) running on the local area network's DHCP 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; it works for both EtherSmart and WiFi Wildcards. For a printed report of the local IP, gateway IP, and netmask, see Ether_IP_Info_Report.

C: xaddr **Ether_Gateway_IP_Ptr** (int modulenum)
4th: **Ether_ Gateway _IP_Ptr** (modulenum -- xaddr)
Returns the xaddress within the ether_info struct for the specified EtherSmart/WiFi module that holds the packed 32-bit IP (Internet Protocol) address of the gateway server (router) on the LAN (Local Area Network). The 4 bytes of the gateway IP address are stored in order at the returned xaddress. Assuming that you have initialized the Ethernet task using Ether_Task_Setup or WiFi_Task_Setup, you can view the gateway IP address that is currently in use by the Lantronix hardware on the EtherSmart/WiFi Wildcard by executing Ether_IP_Info_Request followed by Ether_Await_Response, and then fetching the 4 bytes from the xaddress returned by this routine. For example, if the gateway IP address is 10.0.1.2, the 10 will be stored at the returned xaddress, followed by the 0, the 1, and the 2 in sequential memory bytes. To use a gateway IP address that is automatically assigned by the LAN's gateway host, specify an IP address and gateway IP and netmask of 0.0.0.0 (the factory defaults); then the Lantronix device 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; this works on both the EtherSmart and the WiFi Wildcards. For a printed report of the local IP, gateway IP, and netmask, see Ether_IP_Info_Report. See Ether_Local_IP and Ether_Gateway for more information about how to set the IP address, gateway IP, and netmask.

C: void **Ether_Get_Char**s (xaddr xlbuffer, uint maxbytes, uint maxlines, char eol,
int discard_alt_eol, int discard_msbit_set, uint timeout_msec, int modulenum)
4th: **Ether_Get_Chars** (xlbuf\maxchars\maxlines\eol\discard.alt.eol\no.msbitset\timeout\module--)
Stores the specified input parameters into the ether_info struct and SENDs a message via the ether_command mailbox to the task running the Ether_Service_Loop which dispatches the action function. (If the mailbox is full because the action task hasn't cleared it yet, the SEND routine will PAUSE; see Ether_Ready_For_Command). Requests and stores incoming ASCII data from the specified EtherSmart/WiFi modulenum. The data is stored into xlbuffer starting at the address xlbuffer+2, and the number of bytes received is stored as a 2-byte count at xlbuffer. The xlbuffer parameter is a 32-bit extended address that holds the 16-bit buffer count followed by the buffer data. The data input operation stops if the amount of data in the specified buffer (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 stored into 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 sequences such as CRLF are not allowed. If eol_char = '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 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 tty configuration data is transmitted that can be filtered out by discarding characters with their msbits 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. The specified maxbytes must be less than or equal to 65533 so that the maximum buffersize including count fits in a 16-bit number. After calling this routine the application must clear the ether_response mailbox using Ether_Check_Response or Ether_Await_Response. This routine must not be used to input binary (non-ascii) data; see also Ether_Get_Data and Ether_Add_Data.
Implementation detail: Writes a starting 16-bit count = 0 to the specified xlbuffer and calls Ether_Add_Chars.

C: void **Ether_Get_Data** (xaddr xlbuffer, uint maxbytes, uint timeout_msec, int modulenum)
4th: **Ether_Get_Data** (xlbuffer\maxbytes\timeout_msec\modulenum --)
Stores the specified input parameters into the ether_info struct and SENDs a message via the ether_command mailbox to the task running the Ether_Service_Loop which dispatches the action function. (If the mailbox is full because the action task hasn't cleared it yet, the SEND routine will PAUSE; see Ether_Ready_For_Command). Requests and stores incoming data from the specified EtherSmart/WiFi modulenum. The data is stored into xlbuffer starting at the address xlbuffer+2, and the number of bytes received is stored as a 2-byte count at xlbuffer. The xlbuffer parameter is a 32-bit extended address that holds the 16-bit buffer count followed by the buffer data. The data input operation stops if the amount of data in the specified buffer (not including the 2-byte count) exceeds the specified maxbytes parameter. This function exits within the specified timeout_msec whether or not the maximum number of bytes 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_received 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. The specified maxbytes must be less than or equal to 65533 so that the maximum buffersize including count fits in a 16-bit number. After calling this routine the application must clear the ether_response mailbox using Ether_Check_Response or Ether_Await_Response. See also Ether_Add_Data and Ether_Get_Chars.
Implementation detail: Writes a starting 16-bit count = 0 to the specified xlbuffer and calls Ether_Add_Data.

C: void **Ether_Get_Line** (xaddr xlbuffer, uint maxbytes, char eol, int discard_alt_eol,
int discard_msbit_set, uint timeout_msec, int modulenum)
4th: **Ether_Get_Line** (xlbuffer\maxchars\eol\discard.alt.eol?\no.msbitset?\timeout_msec\module--)
Calls Ether_Get_Chars, passing 1 as the maxlines parameter to input a maximum of one line of text with the specified eol (end of line) character. See Ether_Get_Chars.

C: xaddr **ether_gui_message**
4th: **ether_ gui_message** (-- xaddr)
A mailbox in common RAM that conveys a 32-bit function pointer (xcfa) from the Ethernet control task to the user's application task. The programmer does not directly examine the contents of this mailbox; rather, the programmer's application code monitors the contents of this mailbox using the non-blocking function Ether_Check_GUI function (see its glossary entry for details). When the Ether_Connection_Manager running in the Ether_Service_Loop detects a web request that involves a handler that was posted using HTTP_Add_GUI_Handler, it writes into this mailbox the 32-bit extended code field address of the handler associated with the

incoming web address URL. The User Guide contains information about how to craft handler functions to implement a web-based “remote front panel” for an instrument.
Note: This mailbox is typically not accessed directly. In C, this mailbox can be used as an lvalue or an rvalue, just like any C variable. In Forth, this variable has the standard behavior or returning the xaddress of its contents.

C: xaddr **Ether_Inbuf** (int modulenum)
4th: **Ether_Inbuf** (modulenum -- xaddr)
Returns the 32-bit extended base address of the default buffer for incoming bytes from the Ethernet/WiFi link for the given module. This buffer is available for use by the programmer, and is typically (but not necessarily) passed as the target input buffer to functions such as Ether_Add_Data, Ether_Get_Data, Ether_Add_Chars, Ether_Get_Chars, Ether_Get_Line, and Ether_Add_Line. It can also be used as the “scratchpad buffer” for Ether_Send_Email. Note that these input routines treat this as an “lbuffer” with a 16-bit count stored in the first 2 bytes and the data following. Ether_Inbuf is modified by the following functions that use the Lantronix monitor mode: Ether_XPort_Defaults, Ether_XPort_Update, Ether_Ping_Request, Ether_Ping_Report, Ether_IP_Info_Request, Ether_IP_Info_Response, Ether_Encryption, and WiFi_Encryption. See the glossary entries for Ether_Inbufsize and Ether_Set_Inbuf. Ether_Inbuf should not be used for HTTP (web) service handlers which are dispatched from the Ethernet task; see HTTP_Inbuf.

C: uint **Ether_Inbufsize** (int modulenum)
4th: **Ether_Inbufsize** (modulenum -- u)
Returns the size of the Ether_Inbuf default input buffer for the specified EtherSmart/WiFi module which is set by the Ether_Set_Inbuf function, and/or by Ether_Info_Init, WiFi_Info_Init and their callers as listed in the “Initialization” section of the categorized function list. Note that the allocated buffer size must be 2 bytes bigger than the maxnumbytes parameter passed to Ether_Init, WiFi_Init or Ether_Set_Inbuf; these 2 bytes provide room to store the 16- bit count at the start of the lbuffer. The Ether_Setup and WiFi_Setup functions initialize Ether_Inbufsize to ETHER_BUFSIZE_DEFAULT = 510, with an allocated buffer size of 512 bytes. This is above the recommended minimum ETHER_MIN_BUFFER_SIZE = 320.

C: xaddr **Ether_Info** (int modulenum)
4th: **Ether_Info** (modulenum -- xstruct_base)
Given the module number, fetches the 32-bit extended base address of the ether_info structure as stored by Ether_Info_Init and WiFi_Init and their callers as listed in the “Initialization” section of the categorized function list. There is one ether_info struct per EtherSmart/WiFi Wildcard. The ether_info struct contains the pointers and variables needed to specify the behavior of the corresponding EtherSmart/WiFi Wildcard.

C: void **Ether_Info_Init** (xaddr xinfo_struct_base, xaddr xautoserve_array_base, int numrows,
xaddr xbuffer_area_base, xaddr xcommand_mailbox, xaddr xresponse_mailbox,
xaddr xgui_mailbox, uint ether_inbufsize, uint ether_outbufsize,
uint http_inbufsize, uint http_outbufsize, int modulenum)
4th: **Ether_Info_Init** (xinfo_struct_base\ xautoserve_array_base\numrows
\xbuffer_area_base\xcommand_mailbox\xresponse_mailbox
\xgui_mailbox\ether_inbufsize\ether_outbufsize
\http_inbufsize\http_outbufsize\modulenum --error)

This function initializes an EtherSmart Wildcard; see WiFi_Info_Init to initialize a WiFi Wildcard. Initializes the entries in the ether_info struct located at the specified xinfo_struct_base, and makes a table entry for the specified EtherSmart modulenum/ether_info pair so that the other driver functions can locate the xinfo_struct_base given the modulenum, and passes xautoserve_array_base and numrows to HTTP_Is_Autoserve_Array to allocate the autoserve array, then erases the autoserve array in RAM. Starting at the specified 32-bit xbuffer_area_base location which can be in paged or common RAM, allocates in order Ether_Inbuf having size specified by the ether_inbufsize parameter+2, Ether_Outbuf having size specified by the ether_outbufsize parameter+2, HTTP_Inbuf having size specified by the http_inbufsize parameter+2, and HTTP_Outbuf having size specified by the http_outbufsize parameter+2. The total number of bytes allocated for these buffers is:

8 + ether_inbufsize + ether_outbufsize + http_inbufsize + http_outbufsize

where the 8 + is for the 2-byte count at the beginning of each buffer. In other words, to allow for the 2-byte count to be stored before the content area of each buffer, the allocated size of each buffer is 2 bytes greater than the buffer contents size parameter that is passed to this routine. Stores a 16-bit zero into the first 2 bytes of each of these buffers to initialize each buffer count to zero. This function stores 0\0 into each of the 3 mailboxes ether_command, ether_response, and ether_gui_message specified by xcommand_mailbox, xresponse_mailbox, and xgui_mailbox, respectively. Each mailbox is specified by a 32-bit extended address that points to a 32-bit location in common RAM (mailboxes cannot be located in paged RAM). This function returns a zero flag if there is no error. If Ether_Info_Init detects that the specified size of the Ether_Outbuf or Ether_Inbuf is smaller than ETHER_MIN_BUFFER_SIZE = 320, it returns ERROR_BUFFER_TOO_SMALL = 0x80 and sets Ether_Error to this value (see Ether_Error). This is not a fatal error, but it is best to declare buffers that meet or exceed this minimum size so that all of the functions that rely on these buffers will work properly, including Ether_XPort_Defaults, Ether_XPort_Update, Ether_Ping_Request, Ether_Ping_Report, Ether_IP_Info_Request and Ether_IP_Info_Report. Note that the buffer sizes are available using the functions Ether_Inbuf, Ether_Outbuf, HTTP_Inbuf, and HTTP_Outbuf, and their sizes are available as Ether_Inbufsize, Ether_Outbufsize, HTTP_Inbufsize and HTTP_Outbufsize. This routine turns on the variables pointed to by HTTP_Enable_Ptr and Ether_Tunnel_Enable_Ptr. To disable one or both of these passive services, the program must turn them off explicitly by storing a zero after this routine (or its caller) executes. Initializes the contents of HTTP_Get_Timeout_Msec_Ptr to 5000 (a 5 second timeout for incoming HTTP requests), and initializes HTTP_Timeout_Msec_Ptr to 33000 (a 33 second timeout for outgoing web send operations). Installs the execution xaddress of HTTP_Default_Handler into the HTTP_Default_Handler_Ptr to serve out the “404 Not Found” error page. Sets Ether_Local_Port to its default value of 80 which allows auto-detection of incoming web and passive serial tunneling connections. Sets Ether_Internal_Webserver_Port to its default value of 8000; this is used to access the built-in web configuration tool. Ether_Info_Init is typically called by Ether_Init; see its glossary entry.

C: void **Ether_Init** (xaddr xinfo_struct_base, xaddr xautoserve_array_base, int numrows,
xaddr xbuffer_area_base, xaddr xcommand_mailbox, xaddr xresponse_mailbox,
xaddr xgui_mailbox, uint ether_inbufsize, uint ether_outbufsize,
uint http_inbufsize, uint http_outbufsize, int modulenum)

4th: **Ether_Init** (xinfo_struct_base\ xautoserve_array_base\numrows
\xbuffer_area_base\xcommand_mailbox\xresponse_mailbox
\xgui_mailbox\ether_inbufsize\ether_outbufsize
\http_inbufsize\http_outbufsize\modulenum -- error)

This function initializes an EtherSmart Wildcard; see WiFi_Init to initialize a WiFi Wildcard. This is the fundamental initialization routine that must be invoked before accessing the UART or XPort hardware on the EtherSmart Wildcard. This routine calls Ether_Info_Init; see its glossary entry for a detailed description of its actions including initializing the ether_info struct, mailboxes, timeouts, and local port, and allocating the buffers. Ether_Init initializes the UART chip, powers up the XPort chip, starts the timeslicer and globally enables interrupts. This routine initializes the ether_service_module variable which specifies which module is controlled by the Ether_Service_Loop webserver and command processor running in the Ethernet task. (This routine does not initialize the ether_revector_module variable). This routine flushes the UART input buffers for 0.25 second if no connection is open. If there is an open Ethernet connection, this routine disconnects it, which takes 2.5 seconds. Ether_Init is typically called by Ether_Setup; see its glossary entry.

C: void **Ether_Internal_Webserver_Port** (int portnum, int modulenum)
4th: **Ether_Internal_Webserver_Port** (portnum\modulenum --)

Writes the specified 16-bit internal (configuration) webserver portnum into the ether_info struct for the specified modulenum. This function is typically not used, as the default is typically retained. The default internal webserver port after executing Ether_XPort_Defaults on the EtherSmart/WiFi Wildcard is decimal 8000. To restore the default portnum, pass -1 or decimal 8000 to this function. Assuming that you have initialized the Ethernet task (see Ether_Task_Setup and WiFi_Task_Setup), you can instantiate the values in the Lantronix flash after invoking this function by executing Ether_XPort_Update followed by Ether_Await_Response. To use the web configuration tool, type in the address bar of your browser the IP address of the Wildcard followed by : followed by the specified port number. For example, if the IP address is 10.0.1.22 and the internal webserver port is the default value of 8000, type in the browser address bar:

10.0.1.22:8000

The web configuration tool is self-explanatory, but be aware that changing certain parameters such as the serial configuration (baud rate, etc.) will render this driver code inoperable, and changing the internal webserver port from the web configuration tool itself may cause problems. To recover from such a mishap, initialize the EtherSmart/WiFi Wildcard using Ether_Task_Setup or WiFi_Task_Setup, and then execute Ether_XPort_Defaults followed by Ether_Await_Response. Do not pass the value decimal 9999 to this function, as this is the reserved port for telnet setup and monitor modes. If you specify port 80 as the internal webserver port, be sure to change Ether_Local_Port to a different value.

C: void **Ether_IP_Info_Report** (int modulenum)
4th: **Ether_IP_Info_Report** (modulenum --)

This is an interactive version of Ether_Info_Request, typically typed at a terminal, with the response printed to the terminal. This function retains control for up to approximately 7 seconds (that is, it's a blocking function). This routine invokes the non-blocking function Ether_Info_Request which SENDs a command to the task running the Ether_Service_Loop, which in turn dispatches the action function. (If the mailbox is full, the SEND routine will PAUSE; see Ether_Ready_For_Command). This routine then RECEIVEs the results from the mailbox and prints the info result (along with any appropriate error message) to the terminal. An error code is returned via the ether_response mailbox; a nonzero return value results in a " Couldn't enter monitor mode!" printout. This routine awaits the response for you, so you do not need to clear the ether_response mailbox. The counted response string is at Ether_Outbuf, with the

count in the first 2 bytes and the ascii response string (not including any error messages) following. 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

which summarizes the IP address, gateway address, and netmask, respectively.

Note: The Lantronix firmware may report indeterminate results if DHCP (Dynamic Host Configuration Protocol) is enabled but there is no active network connection. DHCP is enabled by default and after Ether_XPort_Defaults is executed on the EtherSmart/WiFi Wildcard. To disable DHCP, assign a non-zero value to the local IP address, gateway IP, or netmask. See Ether_XPort_Update.

C: void **Ether_IP_Info_Request** (int modulenum)

4th: **Ether_IP_Info_Request** (modulenum --)

SENDs a message via the ether_command mailbox to the task running the Ether_Service_Loop which dispatches the action function. (If the command mailbox is full because the action task hasn't cleared it yet, the SEND routine will PAUSE; see Ether_Ready_For_Command). This function enters monitor mode and retrieves the local IP address, gateway IP address, and netmask that are currently in use by the Lantronix device on the specified EtherSmart/WiFi Wildcard. 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 error flag in the remaining 2 bytes is placed in the ether_response mailbox; the error flag is also available using Ether_Error. The error flag is zero if the operation was successful, or nonzero if we couldn't enter monitor mode. 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 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 (see their glossary entries). The counted ascii response string from the Lantronix device is stored as a 2-byte count followed by the ascii data at Ether_Outbuf. For a printing version, see Ether_Info_Report.

Note: The Lantronix firmware may report indeterminate results if DHCP (Dynamic Host Configuration Protocol) is enabled but there is no active network connection. DHCP is enabled by default and after Ether_XPort_Defaults is executed on the EtherSmart/WiFi Wildcard. To disable DHCP, assign a non-zero value to the local IP address, gateway IP, or netmask. See Ether_XPort_Update.

C: uchar **Ether_Key** (int modulenum)

4th: **Ether_Key** (modulenum -- char)

For the specified EtherSmart/WiFi module, waits for and returns the next pending input character. See Ether_Ascii_Key, Ether_Emit, Ether_Ask_Key, and E_Key.

C: void **Ether_Local_IP** (char ip1, char ip2, char ip3, char ip4, int modulenum)

4th: **Ether_Local_IP** (ip1\ip2\ip3\ip4\modulenum --)

Sets the 32-bit IP (Internet Protocol) address pointed to by Ether_My_IP_Ptr in the ether_info struct to the specified 4-byte value ip1.ip2.ip3.ip4 for the specified EtherSmart/WiFi modulenum. For example, to set the IP address to 10.0.1.22, pass the parameters 10, 0, 1, and 22 followed by the hardware modulenum to this routine. Assuming that you have initialized the Ethernet task (see Ether_Task_Setup and WiFi_Task_Setup), you can implement the specified IP

address so that it is stored in Lantronix flash memory and used by the Lantronix by executing Ether_Local_IP. You should also invoke the Ether_Gateway function to specify the gateway IP address and netmask. Then, to instantiate the values in Lantronix flash, execute Ether_XPort_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 of 0.0.0.0 (the factory default); then 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 on the EtherSmart/WiFi Wildcard. For a printed report of the local IP, gateway IP, and netmask, see Ether_IP_Info_Report. See Ether_Gateway for more information about how to set the gateway IP address and netmask.

C: void **Ether_Local_Port** (int portnum, int modulenum)
4th: **Ether_Local_Port** (portnum\modulenum --)

Writes the specified 16-bit local port number into the ether_info struct for the specified modulenum. The default local port after executing Ether_XPort_Defaults is decimal 80. To confirm or restore the default portnum, pass -1 or decimal 80 to this function. Assuming that you have initialized the Ethernet task (see Ether_Task_Setup and WiFi_Task_Setup), you can instantiate the values in Lantronix flash after invoking this function by executing Ether_XPort_Update followed by Ether_Await_Response. The local port number is the only port on which the Lantronix can receive incoming connections (excluding configuration connections such as setup and monitor mode, and the built-in configuration webserver). Changing the local port value from the default port 80 is not recommended, as it will make using the dynamic webserver more difficult: the end user will have to type in the browser's address bar the :portnum notation after the URL to specify the destination port. Do not pass the value decimal 9999 to this function, as this is the reserved port for telnet setup and monitor modes. Do not pass the value decimal 8000 to this function, as this is the default value for the built-in web configurator tool. If portunm = 0 is specified, then the source port on outgoing (active) connections is a random number between 50000 and 59999.

C: **ETHER_MIN_BUFFER_SIZE**
4th: **ETHER_MIN_BUFFER_SIZE**

A 16-bit constant that returns the value 320. This constant is used by Ether_Init and WiFi_Init to test whether the specified the sizes of the Ether_Outbuf and Ether_Inbuf buffers are big enough to handle the default uses of the buffers for Lantronix configuration via the Ether_XPort_Defaults and Ether_XPort_Update functions. The ETHER_BUFSIZE_DEFAULT constant that is used by Ether_Setup equals 510 bytes, easily passing the constraint. If Ether_Info_Init, Ether_Init, WiFi_Info_Init, or WiFi_Init detects that the specified size of the Ether_Outbuf or Ether_Inbuf is smaller than 320, it sets Ether_Error to ERROR_BUFFER_TOO_SMALL = 0x80 (see Ether_Error). This is not a fatal error, but it is best to declare buffers that meet or exceed this minimum size.

C: void **Ether_Monitor** (void)
4th: **Ether_Monitor** (--)

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/WiFi Wildcard whose modulenum is stored in the ether_revector_module global variable. Make sure that you initialize

ether_revector_module before executing this function or using this function as a task activation routine via the ACTIVATE kernel function.
Usage notes: See the demo program for an example of use. Store the correct modulenum that corresponds to the hardware jumper settings of the EtherSmart/WiFi Wildcard into the global variable named ether_revector_module, then call Ether_Monitor. This will run the default (startup) task through the specified EtherSmart/WiFi wildcard. To communicate with the terminal, use the "Rlogin" mode of the Putty TCP/IP terminal program to connect to the local port (typically = 80) at the specified EtherSmart/WiFi IP address (see Ether_Local_IP), and you're talking to the QED monitor via 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 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/WiFi Wildcard, then build and activate a task using Ether_Monitor as the activation routine as shown in the demo code. Use Putty "Rlogin" to connect to the local port (typically port 80) at the specified EtherSmart/WiFi IP address, and you're talking to the task via Ethernet/WiFi.
To obtain the free Putty TCP/IP terminal program, type

Putty

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

C: xaddr **Ether_My_IP_Ptr** (int modulenum)
4th: **Ether_My_IP_Ptr** (modulenum -- xaddr)
Returns the xaddress within the ether_info struct for the specified EtherSmart/WiFi module that holds the packed 32-bit IP (Internet Protocol) address of the Lantronix device. The IP address is used by the network to uniquely identify the EtherSmart/WiFi Wildcard, and consequently the IP address must be unique on the Local Area Network (LAN) containing the EtherSmart/WiFi Wildcard. The 4 bytes of the IP address are stored in order at the returned xaddress. Assuming that you have initialized the Ethernet task using Ether_Task_Setup or WiFi_Task_Setup, you can view the IP address that is currently in use by the Lantronix hardware on the EtherSmart/WiFi Wildcard by executing Ether_IP_Info_Request followed by Ether_Await_Response, and then fetching the 4 bytes from the xaddress returned by this routine. For example, if the IP address is 10.0.1.22, the 10 will be stored at the returned xaddress, followed by the 0, the 1, and the 22 in sequential memory bytes. To use an IP address that is automatically assigned by the LAN's gateway host, specify an IP address of 0.0.0.0 (the factory default); then 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. For a printed report of the local IP, gateway IP, and netmask, see Ether_IP_Info_Report. See Ether_Local_IP and Ether_Gateway for more information about how to set the IP address, gateway IP, and netmask.

C: xaddr **Ether_Netmask_Ptr** (int modulenum)
4th: **Ether_ Netmask_Ptr** (modulenum -- xaddr)
Returns the xaddress within the ether_info struct for the specified EtherSmart/WiFi module that holds the packed 32bit netmask of the LAN (Local Area Network). The 4 bytes of the netmask are stored in order at the returned xaddress. Assuming that you have initialized the Ethernet task using Ether_Task_Setup or WiFi_Task_Setup, you can view the netmask that is currently in use by the Lantronix hardware on the EtherSmart/WiFi Wildcard by executing

Ether_IP_Info_Request followed by Ether_Await_Response, and then fetching the 4 bytes from the xaddress returned by this routine. For example, if the netmask is 255.255.255.0, the first 255 will be stored at the returned xaddress, followed by 255, 255, and the 0 in sequential memory bytes (this example corresponds to 8 "subnet bits"; see Ether_Gateway). To use a netmask that is automatically assigned by the LAN's gateway host, specify an IP address and gateway IP and netmask of 0.0.0.0 (the factory defaults); then the Lantronix hardware relies on DHCP (Dynamic Host Configuration Protocol) running on the local area network's gateway. To revert to the factory defaults which rely on a DHCP-assigned IP, gateway IP and netmask, execute Ether_XPort_Defaults. For a printed report of the local IP, gateway IP, and netmask, see Ether_IP_Info_Report. See Ether_Local_IP and Ether_Gateway for more information about how to set the IP address, gateway IP, and netmask.

C: xaddr **Ether_Outbuf** (int modulenum)
4th: **Ether_Outbuf** (modulenum -- xaddr)
Returns the 32-bit extended base address of the default buffer for outgoing bytes to the Ethernet/WiFi link for the given module. This buffer is available for use by the programmer, and is typically (but not necessarily) passed as the target output buffer to functions such as Ether_Send_Buffer, Ether_Send_LBuffer, Ether_Send_2Buffers, and Ether_Send_Email. Note that Ether_Send_LBuffer expects to find the 16-bit count stored in the first 2 bytes of the buffer, with the data following. Ether_Outbuf is modified by functions that use the Lantronix monitor mode: Ether_XPort_Defaults, Ether_XPort_Update, Ether_Ping_Request, Ether_Ping_Report, Ether_IP_Info_Request, Ether_IP_Info_Response, Ether_Encryption, and WiFi_Encryption. See also Ether_Outbufsize and Ether_Set_Outbuf. The convenient Ether_Outbuf_Cat function adds a specified string to Ether_Outbuf. Ether_Outbuf should not be used for HTTP (web) service handlers which are dispatched from the Ethernet task; see HTTP_Outbuf.

C: void **Ether_Outbuf_Cat** (xaddr xstring, uint count, int modulenum)
4th: **Ether_Outbuf_Cat** (xstring\count\modulenum --)
Concatenates the specified string plus a terminating carriage return and linefeed to the specified module's Ether_Outbuf buffer, and increments the buffer count that is stored in the first two bytes of Ether_Outbuf to reflect the updated buffer contents. The string is specified by the extended 32-bit address xstring of its first character, and has count bytes to be added to the Ether_Outbuf buffer. An end-of-line sequence equal to 0x0D0A (carriage return and linefeed) is post-pended to the Ether_Outbuf after the string is added. This routine clamps the maximum number of bytes in the destination buffer to the value returned by Ether_Outbufsize (this value does not include the 2-byte count stored at the start of Ether_Outbuf). This function is useful for building up strings to be sent out via TCP/IP connections.
NOTE: Before adding the first string, be sure to store a 16-bit zero into the xaddress returned by Ether_Outbuf for the specified module; this initializes the counted lbuffer to its starting size of zero.

C: uint **Ether_Outbufsize** (int modulenum)
4th: **Ether_Outbufsize** (modulenum -- u)
Returns the size of the Ether_Outbuf default output buffer for the specified EtherSmart/WiFi module which is set by the Ether_Set_Outbuf function, and/or by Ether_Info_Init, WiFi_Info_Init, and its callers as listed in the "Initialization" section of the categorized function list. Note that the allocated buffer size must be 2 bytes bigger than the maxnumbytes parameter passed to Ether_Init or Ether_Set_Outbuf; these 2 bytes provide room to store the 16-bit count at the start of the lbuffer. To guarantee that Ether_Outbuf will be able to hold s-records as required for the

monitor mode commands such as Ether_XPort_Defaults and Ether_XPort_Update, Ether_Outbufsize should be initialized to a value greater than or equal to ETHER_MIN_BUFFER_SIZE = 320. The Ether_Setup and WiFi_Setup functions initialize Ether_Outbufsize to ETHER_BUFSIZE_DEFAULT = 510, with an allocated buffer size of 512 bytes.

C: int **Ether_Passive_Non_Web_Connection** (int modulenum)
4th: **Ether_Passive_Non_Web_Connection** (modulenum -- flag)

Returns a true flag if a passive connection has been accepted that did not identify as HTTP (that is, there was no leading GET keyword detected). The main application task loop can repeatedly call this routine to see if an incoming "serial tunneling" connection needs to be serviced.

Implementation detail: Returns a true (nonzero) flag if Ether_Connect_Status returns the value 6; otherwise returns false. See the glossary entry for Ether_Connect_Status for a detailed description.

C: void **Ether_Ping_Report** (char ip1, char ip2, char ip3, char ip4, int modulenum)
4th: **Ether_Ping_Report** (ip1\ip2\ip3\ip4\modulenum --)

This is an interactive version of ping, typically typed at a terminal, with the response printed to the terminal. This function retains control for up to approximately 13 seconds (that is, it's a blocking function). For a non-blocking version, use Ether_Ping_Request. A "ping" is a way of finding out whether a remote host is available on the network; if so, the remote responds to the ping. This routine invokes Ether_Ping_Request which writes the specified input parameters into the ether_info struct, and SENDs to the task running the Ether_Service_Loop which dispatches the action function to send an outgoing ping from the Lantronix device to a device with IP address ip1.ip2.ip3.ip4. (If the mailbox is full, the SEND routine will PAUSE; see Ether_Ready_For_Command). This routine then RECEIVEs the results from the mailbox and prints the ping result (along with any appropriate error message) to the terminal. An error code is returned via the ether_response mailbox; a nonzero return value results in a " Couldn't enter monitor mode!" or " No response from remote" printout. This routine awaits the response for you, so you do not need to clear the ether_response mailbox. The counted ping response string is at Ether_Outbuf, with the count in the first 2 bytes and the ascii response string (not including error messages) following. If the remote responded to the ping, the ping report is of the form:

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

Of course, the reported time will vary depending on how long it took the remote to respond to each ping.

C: void **Ether_Ping_Request** (char ip1, char ip2, char ip3, char ip4, int modulenum)
4th: **Ether_Ping_Request** (ip1\ip2\ip3\ip4\modulenum --)

Stores the specified input parameters into the ether_info struct and SENDs a message via the ether_command mailbox to the task running the Ether_Service_Loop which dispatches the action function, with an error flag placed in Ether_Error and in the ether_response mailbox to indicate whether the remote responded as described below. (If the command mailbox is full because the action task hasn't cleared it yet, the SEND routine will PAUSE; see Ether_Ready_For_Command). A "ping" is a way of finding out whether a remote host is

available on the network; if so, the remote responds to the ping. This function enters monitor mode and sends an outgoing ping from the Lantronix device to a device with IP address ip1.ip2.ip3.ip4, waits for return string, and verifies that it has enough bytes to indicate that the remote responded to the ping. The captured ping results are placed as a packed lstring at Ether_Outbuf with the count in first 2 bytes and the ascii ping response following (see the glossary entry of Ether_Ping_Report for the format of the response). The remote ping response comprises 6 lines (1 per second), or nothing if the remote did not respond. The entire operation takes approximately 13 seconds. 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 error flag in the remaining 2 bytes is placed in the ether_response mailbox; the error flag is also available using Ether_Error. The error flag is zero if the operation was successful, or 0x08 (ERROR_INVALID_RESPONSE; see Ether_Error) if we couldn't enter monitor mode, or 0x10 (ERROR_NO_RESPONSE; see Ether_Error) if the ping was sent but the remote did not respond. 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. See also Ether_Ping_Report.

C: int **Ether_Ready_For_Command** (int modulenum)
4th: **Ether_Ready_For_Command**(modulenum -- ready?)
Checks the state of the ether_command mailbox for the specified module and returns a true flag if the mailbox is empty (i.e, if its contents equal 0\0). Returns a false flag if the contents of ether_command are non-zero, indicating that the mailbox is full and not ready to receive another command. The application program can use this function to avoid invoking a command until the Ethernet task is done receiving the prior command; this can avoid potentially long pauses if several commands have been issued, as the SEND routine invoked by the Ethernet command functions will loop and call Pause if the mailbox is still full. See the glossary entry for Ether_Command_Manager for a summary of the relevant commands that write to the ether_command mailbox.
To use: call Ether_Ready_For_Command before invoking one of the commands that writes to the command mailbox, and execute the command only if Ether_Ready_For_Command returns true.

C: xaddr **Ether_Remote_IP_Ptr** (int modulenum)
4th: **Ether_Remote_IP_Ptr** (module -- xaddr)
Returns the xaddress within the ether_info struct for the specified EtherSmart/WiFi module that holds the packed 32-bit IP (Internet Protocol) address of the remote computer that was the subject of the last outgoing connection or ping request. This pointer is typically used only for diagnostic purposes, as the remote IP is fully specified in the parameter list that is passed to the functions that initiate the outgoing connection requests.

C: xaddr **Ether_Remote_Port_Ptr** (int modulenum)
4th: **Ether_Remote_Port_Ptr** (module -- xaddr)
Returns the xaddress within the ether_info struct for the specified EtherSmart/WiFi module that holds the 16-bit port of the remote computer that was the subject of the last outgoing connection or ping request. This pointer is typically used only for diagnostic purposes, as the remote port is specified in the parameter list that is passed to the functions that initiate the outgoing connection requests.

C: ulong **ether_response**
4th: **ether_ response** (-- xaddr)
A mailbox in common RAM that conveys a 32-bit value from the Ethernet control task to the user's application task. The programmer monitors the contents of this mailbox using either the non-blocking function Ether_Check_Response, or the blocking function Ether_Await_Response; see their glossary entries for details. When the Ether_Command_Manager running in the Ether_Service_Loop finishes executing a command that is dispatched from the user's application task, it writes into this mailbox a command_id byte and modulenum in the msword, and an optional 16-bit return parameter in the lsword. The return parameter may be an error flag, the number of bytes transmitted by a send command, etc.
Note: This mailbox is typically not accessed directly. In C, this mailbox can be used as an lvalue or an rvalue, just like any C variable. In Forth, this variable has the standard behavior or returning the xaddress of its contents.

C: int **ether_revector_module**
4th: **ether_revector_module** (-- xaddr)
A 16-bit variable in common RAM containing a modulenum. This variable must be initialized before calling Ether_Serial_Revector or Ether_Monitor to revector serial I/O via the EtherSmart/WiFi Wildcard network port. This variable is not initialized by default. The contents of this variable are used by E_Key, E_ASCII_Key, E_Ask_Key, and E_Emit to implement revectored serial. The specified value must match the hardware jumper setting on the Wildcard that is to be used for revectored I/O. For example, if both of the module ID jumpers are installed and the Wildcard is mounted on Wildcard Module Header 0, then the value 3 should be stored into the variable to access the Wildcard if you want to download code through this Wildcard to the controller. See the User Guide for details about setting the module number. See the glossary entries for Ether_Serial_Revector and Ether_Monitor. The Ethernet/WiFi Demo code provided in source form presents an example of how to revector downloads and interactive program development through the EtherSmart/WiFi Wildcard.
Note: In C, this variable can be used as an lvalue or an rvalue, just like any C variable. In Forth, this variable has the standard behavior or returning the xaddress of its contents.

C: void **Ether_Send_2Buffers** (xaddr xbuffer1, uint count1, xaddr xbuffer2, uint count2,
uint timeout_msec, int modulenum)
4th: **Ether_Send_2Buffers** (xbuffer1\count1\xbuffer2\count2\timeout_msec\modulenum--)
Stores the specified input parameters into the ether_info struct and SENDs a message via the ether_command mailbox to the task running the Ether_Service_Loop which dispatches the action function. (If the mailbox is full because the action task hasn't cleared it yet, the SEND routine will PAUSE; see Ether_Ready_For_Command). Sends to the EtherSmart/WiFi Wildcard up to count1 bytes of data starting at the 32-bit extended memory address xbuffer1, and then sends up to count2 bytes of data starting at the 32-bit extended memory address xbuffer2. Does not send bytes if there is no connection, or if there has been a change in connection status (e.g., a transient disconnect) as detected by Ether_Disconnect_During_Send; in these cases, this routine increments the connection status value returned by Ether_Connect_Status to an odd value to flag the transient disconnect event. Any required end of line characters such as carriage return (0x0D) and linefeed (0x0A) characters must be in the buffers; they are not added by this routine. Each of the two send operations exits within the specified timeout_msec whether or not the maximum number of bytes have been sent. 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 the total numbytes_sent in the remaining 2

bytes is placed in the ether_response mailbox. After calling this routine the application must clear the ether_response mailbox using Ether_Check_Response or Ether_Await_Response. The total number of bytes actually sent is also available via Ether_Numbytes_Sent. See also HTTP_Send_2Buffers, HTTP_GUI_Send_2Buffers, Ether_Send_Buffer and Ether_Send_Buffer.

C: void **Ether_Send_Buffer** (xaddr xbuffer, uint count, uint timeout_msec, int modulenum)
4th: **Ether_Send_Buffer** (xbuffer\count\timeout_msec\modulenum --)
Stores the specified input parameters into the ether_info struct and SENDs a message via the ether_command mailbox to the task running the Ether_Service_Loop which dispatches the action function. (If the mailbox is full because the action task hasn't cleared it yet, the SEND routine will PAUSE; see Ether_Ready_For_Command). Sends to the EtherSmart/WiFi Wildcard up to count bytes of data starting at the 32-bit extended memory address xbuffer. Does not send bytes if there is no connection, or if there has been a change in connection status (e.g., a transient disconnect) as detected by Ether_Disconnect_During_Send; in these cases, this routine increments the connection status value returned by Ether_Connect_Status to an odd value to flag the transient disconnect event. Any required end of line characters such as carriage return (0x0D) and linefeed (0x0A) characters must be in the buffer; they are not added by this routine. This function exits within the specified timeout_msec whether or not the maximum number of bytes have been sent. 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_sent in the remaining 2 bytes is placed in the ether_response mailbox. After calling this routine the application must clear the ether_response mailbox using Ether_Check_Response or Ether_Await_Response. The number of bytes actually sent is also available via Ether_Numbytes_Sent. See also HTTP_Send_Buffer, HTTP_GUI_Send_Buffer, Ether_Send_LBuffer and Ether_Send_2Buffers.

C: void **Ether_Send_Email** (xaddr xemail_body, uint body_cnt, xaddr xhostname, int host_cnt,
xaddr xsender, int sender_cnt, xaddr xreceiver, int receiver_cnt, xaddr xscratchbuf,
char dest_ip1, char dest_ip2, char dest_ip3, char dest_ip4, int dest_port,
uint timeout_msec, int modulenum)
4th: **Ether_Send_Email** (xemail_body \body_cnt\xhostname \host_cnt
\xsender \sender_cnt\ xreceiver\receiver_cnt\xscratch_buf
\dest_ip1\dest_ip2\dest_ip3\dest_ip4\dest_port\timeout_msec\modulenum --)
Stores the specified input parameters into the ether_info struct of the specified EtherSmart/WiFi modulenum and SENDs a message via the ether_command mailbox to the task running the Ether_Service_Loop which dispatches the action function. (If the mailbox is full because the action task hasn't cleared it yet, the SEND routine will PAUSE; see Ether_Ready_For_Command). This function sends an email to the remote host at IP (Internet Protocol) address dest_ip1.dest_ip2.dest_ip3.dest_ip4 and port dest_port. The highly recommended dest_port is port 25, which is the SMTP (Simple Mail Transfer Protocol) email well-known port. This function allows the Lantronix device to be an SMTP client that initiates an outgoing email, sending it to a remote host that is running an SMTP daemon (mail server). This implementation of SMTP opens a connection to the server, sends a single email, quits and closes the connection. While the specified IP and port must be on the Local Area Network, sending an email to a non-local recipient can be accomplished by configuring the gateway router on your LAN to relay email messages to the non-local recipient (contact your LAN system administrator for assistance if this is required). The specified scratchpad RAM buffer xscratch_buf holds temporary data going to and from the Lantronix device; it must be at least 16 bytes longer than each specified string input buffer (except for the body string, which is never

placed into the scratch buffer). Ether_Inbuf provides a convenient scratchpad buffer of adequate size; feel free to pass Ether_Inbuf as the xscratch_buf if this is compatible with your application. The timeout_msec function sets the maximum time for each phase of the email data exchange process to complete; make sure that it is long enough to deal with any delays in your LAN; setting the delay to 10000 (10 seconds) is a reasonable default. The remaining input parameters specify the starting xaddresses and counts of the component strings in the email header and body; these buffers can be in RAM, or in flash memory if their contents are static and pre-initialized. Each xaddress (32-bit extended address) points to the first character in the string, and each corresponding count specifies the number of characters in the string. The xemail_body and body_cnt parameters specify the body of the email. The email body is sent without transferring it to a temporary buffer to improve efficiency and to allow static email bodies to be sent directly from flash memory to the remote. (The transfer is a bit tricky, however, as this routine implements the “transparency” requirement specified by the email protocol; experts can consult RFC821, section 4.5.2 available on the web to find out how it’s done). To build up a dynamic email body “on the fly”, consider using the convenient function Ether_Outbuf_Cat to construct the dynamic email text. Just make sure that the result fits in the specified Ether_Outbuf_Size, and remember to pass Ether_Outbuf+2 as the address of the first character, and the 16-bit contents of Ether_Outbuf as the body_cnt (because Ether_Outbuf contains a 2-byte count followed by the buffer contents). The hostname xaddress and count specify a string that identifies the computer that originates the email; for example:

" wildcard.mosaic-industries.com"

The sender xaddress and count specify a string that identifies the sender of the email; for example:

" niceguy@gateway.mosaic-industries.com"

The receiver xaddress and count specify a string that identifies the recepient of the email; for example:

" notso_niceguy@gateway.mosaic-industries.com"

This Ether_Send_Email function automatically emplaces a To: header line in the email referencing the recipient; this field is sent separately via the scratchpad buffer and does not consume any room in the xemail_body buffer. Other optional fields can be explicitly placed by your application program inside the body text. One that is very useful is the subject line, which is emplaced in the body as a line of the form:

" Subject: Wildcard Email Test-Drive "

The word Subject: should be the first word on the line, and put a space after the colon, with a carriage return/linefeed (0x0D0A) terminating the line (this is the standard line termination for email messages). Other optional fields of similar form include

Date:
Cc:
From:

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 error code (also available in Ether_Error) in the remaining 2 bytes is placed in the ether_response mailbox.

The numeric (unnamed) error codes are as follows:

Value	Meaning	
0x00	no error; operation was successful	
0x10	ERROR_NO_RESPONSE	// no response from remote
0x20	ERROR_ALREADY_CONNECTED	// only 1 connection is allowed at a time
0x100	ERROR_TIMEOUT	// timed out before operation could complete

Other SMTP-related error codes:
251 User not local; will forward to <forward-path>
551 User not local; please try <forward-path>
500 Line too long.
501 Path too long
552 Too many recipients.
552 Too much mail data.
After calling this routine the application must clear the ether_response mailbox using Ether_Check_Response or Ether_Await_Response.
Note: The demo code presented in source form presents an example of how to use this function.

C: void **Ether_Send_LBuffer** (xaddr xlbuffer, uint timeout_msec, int modulenum)
4th: **Ether_Send_LBuffer** (xlbuffer\timeout_msec\modulenum --)
Stores the specified input parameters into the ether_info struct and SENDs a message via the ether_command mailbox to the task running the Ether_Service_Loop which dispatches the action function. (If the mailbox is full because the action task hasn't cleared it yet, the SEND routine will PAUSE; see Ether_Ready_For_Command). The xlbuffer parameter is a 32-bit extended address that holds the 16-bit buffer count followed by the buffer data. Fetches the count from xlbuffer and sends to the EtherSmart/WiFi Wildcard up to count bytes of data that are located at xlbuffer+2. Does not send bytes if there is no connection, or if there has been a change in connection status (e.g., a transient disconnect) as detected by Ether_Disconnect_During_Send; in these cases, this routine increments the connection status value returned by Ether_Connect_Status to an odd value to flag the transient disconnect event. Any required end of line characters such as carriage return (0x0D) and linefeed (0x0A) characters must be in the buffer; they are not added by this routine. This function exits within the specified timeout_msec whether or not the maximum number of bytes have been sent. 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_sent in the remaining 2 bytes is placed in the ether_response mailbox. After calling this routine the application must clear the ether_response mailbox using Ether_Check_Response or Ether_Await_Response. The number of bytes actually sent is also available via Ether_Numbytes_Sent. See also HTTP_Send_LBuffer, HTTP_GUI_Send_LBuffer, Ether_Send_Buffer and Ether_Send_2Buffers.

C: void **Ether_Serial_Revector** (void)
4th: **Ether_Serial_Revector** (--)
For the current task (that is, the task that invokes this routine), revectors the serial primitives to use the EtherSmart/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 this routine, as the variable is not initialized by default. See Ether_Monitor.
Implementation detail: Stores the xcfa (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_Key into UKEY. To restore the standard RS232 serial ports, execute COLD. (If there was the standard SAVE at the end of your program, executing RESTORE after the COLD will restore access to the defined functions in your application).

C: void **Ether_Service_Loop** (void)

4th: **Ether_Service_Loop** (--)
An infinite loop task activation routine that repeatedly calls Ether_Connection_Manager and Ether_Command_Manager, passing as the specified modulenum the contents of the ether_service_module variable. See Ether_Task_Setup and WiFi_Task_Setup.

C: int **ether_service_module**
4th: **ether_service_module** (-- xaddr)
A 16-bit variable in common RAM containing a modulenum. This variable is initialized by Ether_Init, WiFi_Init and their calling functions (listed in the "Initialization" section of the categorized function list) to specify which Wildcard module is accessed by the Ether_Service_Loop routine running in the Ethernet control task. The specified value must match the hardware jumper setting on the Wildcard. For example, if neither of the module ID jumpers are installed and the Wildcard is mounted on Wildcard Module Header 0, then the value 0 should be stored into the variable to access the Wildcard. See the User Guide for details about setting the module number. See the Ethernet/WiFi Demo code provided in source form for an example of how to setup the Ethernet control task.
Note: In C, this variable can be used as an lvalue or an rvalue, just like any C variable. In Forth, this variable has the standard behavior or returning the xaddress of its contents.

C: void **Ether_Set_Inbuf** (xaddr xbufbase, int maxnumbytes, int modulenum)
4th: **Ether_Set_Inbuf** (xbufbase\maxnumbytes\modulenum --)
Sets the 32-bit extended base address xbufbase and the size maxnumbytes of the Ether_Inbuf default input buffer. Note that the allocated buffer size must be 2 bytes bigger than the maxnumbytes parameter; these 2 bytes provide room to store the 16- bit count at the start of the lbuffer. The Ether_Setup and WiFi_Setup functions initialize Ether_Inbufsize to ETHER_BUFSIZE_DEFAULT = 510, with an allocated buffer size of 512 bytes.

C: void **Ether_Set_Outbuf** (xaddr xbufbase, int maxnumbytes, int modulenum)
4th: **Ether_Set_Outbuf** (xbufbase \maxnumbytes\modulenum --)
Sets the 32-bit extended base address xbufbase and the size maxnumbytes of the Ether_Outbuf default output buffer. Note that the allocated buffer size must be 2 bytes bigger than the maxnumbytes parameter; these 2 bytes provide room to store the 16- bit count at the start of the lbuffer. To guarantee that Ether_Outbuf will be able to hold s-records as required for the monitor mode commands Ether_XPort_Defaults and Ether_XPort_Update, Ether_Outbufsize should be initialized to a value greater than or equal to ETHER_MIN_BUFFER_SIZE = 320. The Ether_Setup and WiFi_Setup functions initialize Ether_Outbufsize to ETHER_BUFSIZE_DEFAULT = 510, with an allocated buffer size of 512 bytes.

C: uint **Ether_Setup** (xaddr xbuffer_area_base, addr mailbox_base_addr, int modulenum)
4th: **Ether_Setup** (xbuffer_area_base\ mailbox_base_addr \module -- numbytes)
This function initializes an EtherSmart Wildcard; see WiFi_Setuup to initialize a WiFi Wildcard. Ether_Setup is a high level initialization routine that calls Ether_Init with default buffer locations and sizes starting at the specified xbuffer_area_base, and returns the number of bytes allocated there. Also passes to Ether_Init the addresses of three mailboxes in common RAM starting at mailbox_base (a 16-bit address). The mailboxes are ether_command at mailbox_base_addr, ether_response at mailbox_base_addr+4, and ether_gui_message at mailbox_base_addr+8. These mailboxes must be declared (allocated) before calling this function. Locates all required buffers, the ether_info struct, and the HTTP autoserve array in RAM starting at the specified

xbuffer_area_base, and returns the number of bytes allocated. Note that xbuffer_area_base can be in paged RAM to conserve common RAM for other uses. Zeros the three required mailboxes ether_command, ether_response, and ether_gui_message in common RAM. Starting at the specified 32-bit xbuffer_area_base, allocates the ether_info struct, followed in order by the autoserve array, Ether_Inbuf (512 bytes including 2-byte count), Ether_Outbuf (512 bytes including 2-byte count), HTTP_Inbuf (256 bytes including 2-byte count), and HTTP_Outbuf (1024 bytes including 2-byte count). Initializes the UART chip, powers up the XPort chip, starts the timeslicer and globally enables interrupts. This routine initializes the ether_service_module variable which specifies which module is controlled by the Ether_Service_Loop webserver & command processor running in the Ethernet task. (This routine does not initialize the ether_revector_module variable). This routine flushes the UART input buffers for 0.25 second if no connection is open. If there is an open Ethernet connection, this routine disconnects it, which takes 2.5 seconds. Ether_Setup is typically called by Ether_Setup_Default; see its glossary entry.

C: uint **Ether_Setup_Default** (int modulenum)
4th: **Ether_Setup_Default** (module -- numbytes)

This function initializes an EtherSmart Wildcard; see WiFi_Setup_Default to initialize a WiFi Wildcard. Ether_Setup_Default is a high level initialization routine that calls Ether_Setup with a default xbuffer_area_base. For controllers running V6.xx kernels, passes 0x178000 (address 0x8000 on page 0x17) as the xbuffer_area_base to Ether_Setup. For controllers running V4.xx kernels, passes 0x034000 (address 0x4000 on page 3) as the xbuffer_area_base to Ether_Setup. Returns the number of bytes allocated at xbuffer_area_base; it is less than 3 Kbytes. See Ether_Setup for a detailed description of operation. Ether_Setup_Default is typically called by Ether_Task_Setup; see its glossary entry.

Note: Ether_Setup_Default specifies the same memory map as WiFi_Setup_Default; these functions cannot be invoked for more that one Wildcard on a single controller. If your application requires multiple EtherSmart or WiFi Wildcards (or some combination or the two), you must specify non-overlapping memory maps and use Ether_Setup or a lower level initialization function to configure each Wildcard.

C: void **Ether_Shutdown** (int modulenum)
4th: **Ether_Shutdown** (modulenum --)

Shuts down the linear 3.3 volt regulator on the EtherSmart/WiFi Wildcard, thus saving over 1.25W = 0.25A at 5V. 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 the “Initialization” section of the categorized funtion list.

C: void **Ether_Task_Setup** (TASK* task_base_addr, int modulenum)
4th: **Ether_Task_Setup** (task_base_addr\modulenum --)

This function initializes an EtherSmart Wildcard; see WiFi_Task_Setup to initialize a WiFi Wildcard. Ether_Task_Setup is a high level routine that performs a full initialization of the ether_info struct and mailboxes, and builds and activates an Ethernet control task to service the XPort for the specified modulenum. Calls Ether_Setup_Default which in turn calls Ether_Setup with a default xbuffer_area_base. For controllers running V6.xx kernels, passes 0x178000 (address 0x8000 on page 0x17) as the xbuffer_area_base to Ether_Setup. For controllers running V4.xx kernels, passes 0x034000 (address 0x4000 on page 3) as the xbuffer_area_base to Ether_Setup. Less than 3 Kbytes is allocated in this buffer area. See Ether_Setup for a

detailed description of operation. Ether_Task_Setup then builds a standard task running the Ether_Service_Loop activation routine at the specified 16-bit task_base address in common RAM (task areas must be in common RAM). To use this routine, allocate a 1 Kbyte (1024 byte) task area in common RAM (see the demo program described in the User Guide for an example; GCC task areas are allocated by the linker and may be larger than 1K). Pass its 16-bit base address along with the EtherSmart Wildcard modulenum (that corresponds to the hardware jumper settings) to this routine to perform a complete initialization and start the Ethernet task to service the connections. See the demo code for a convenient Ether_Task_Setup_Default routine that uses a pre-defined 1 Kbyte task area to call this function. Also see the demo code for a Web_Demo function that hosts a demonstration website from the EtherSmart Wildcard.
Note: Ether_Task_Setup specifies the same memory map as WiFi_Task_Setup; these functions cannot be invoked for more that one Wildcard on a single controller. If your application requires multiple EtherSmart or WiFi Wildcards (or some combination or the two), you must specify non-overlapping memory maps and use Ether_Setup or a lower level initialization function to configure each Wildcard.

C: void **Ether_TCP_Control** (char keepalive_sec, char discon_minutes,
int send_imm_chars, char flushmode, char packing, int module)

4th: **Ether_TCP_Control** (keepalive_sec\discon_min\send_imm_chars\flush\packing\module--)

This function is for experts only. Maintaining the default values as set by Ether_XPort_Defaults is highly recommended. This function writes the specified parameters into the ether_info struct for the specified module. Keepalive_sec specifies the number of seconds between “keepalive” signals sent on the network; the default 45, and the valid range is 1 to 65, with a value of 255 disabling the keepalive feature. The discon_minutes parameter forces a disconnect after the specified number of minutes of inactivity; 0-99 is the valid range, with 0 (the default) meaning that the inactivity timeout feature is disabled. The send_imm_chars parameter specifies a 1- or 2-character sequence that, when encountered, triggers an immediate send to the network. if the most significant (ms) byte of this parameter is 0, the feature is disabled. If the most significant (ms) byte is nonzero and the least significant (ls) byte is zero, the single ms byte specifies the send character. If both bytes are nonzero and the packing parameter specifies a 2-byte send sequence, then the 2-byte sequence must be encountered to trigger the immediate send. The flushmode specifies when the Lantronix device’s input and output buffer contents are flushed (discarded); the default value of 0x80 (no flushing, pack control enabled) is highly recommended to avoid loss of data while ensuring fast network response over a wide range of conditions. Other flushmode values are bitmapped as follows (bitmaps can be OR’d to achieve flushing combinations):

0x01 = network->serial clear with active connect
0x02 = network -> serial clear with passive connect through network
0x04 = network -> serial clear at time of disconnect
0x10 = serial -> network clear with active connect (initiated by us)
0x20 = serial -> network clear with passive connect (connect request from network)
0x40 = serial -> network clear at time of disconnect
0x80 = alternate packing algorithm

The packing parameter has a recommended default value of zero; the meaning of this byte is as follows:

0x00 = Idle time to force transmit: 12 ms (default)
0x01 = Idle time to force transmit: 52 ms
0x02 = Idle time to force transmit: 250 ms
0x03 = Idle time to force transmit: 5 s

0x00 = No trailing chars after sendchar(s)
0x04 = One trailing char after sendchar(s)
0x08 = Two trailing chars after sendchar(s)
0x10 = Sendchar define 2-byte sequence
0x20 = Send immediate after sendchar

After executing this function, assuming that you have initialized the Ethernet task (see Ether_Task_Setup or WiFi_Task_Setup), you can instantiate the values into the Lantronix flash by executing Ether_XPort_Update followed by Ether_Await_Response.

C: void **Ether_Telnet_Password** (xaddr xpassword_string, int password_cnt, int modulenum)
4th: **Ether_Telnet_Password**(xpassword_string\count\modulenum --)

Writes the specified string specifier parameters into the ether_info struct for the specified module as the password used by the telnet configuration modes. The xaddr xpassword_string is the 32-bit base address of the first character of the string, and count is the number of bytes in the string (clamped to a maximum of 4 bytes). Assuming that you have initialized the Ethernet task (see Ether_Task_Setup or WiFi_Task_Setup), you can instantiate the string into the Lantronix flash after invoking this function by executing Ether_XPort_Update followed by Ether_Await_Response. Use of this string is optional; it prohibits access to the setup mode via telnet to port 9999 unless the proper password is entered. To revert to the default (no password), use Ether_XPort_Defaults.

NOTE: If you set a telnet password, the web interface tool may not allow a login if you have specified non-zero parameters for the Ether_Gateway function. This seems to be a bug in the Lantronix web configuration tool. The telnet configuration tools work under these conditions.

C: xaddr **Ether_Tunnel_Enable_Ptr** (int modulenum)
4th: **Ether_Tunnel_Enable_Ptr** (modulenum -- xaddr)

Returns the xaddress of a a 16-bit flag within the ether_info struct for the specified EtherSmart/WiFi module. If the flag is false (zero), the Ether_Command_Manager will close any incoming connection that it identifies as non-http (because a leading GET substring is not found). The default value of this flag is true: passive serial tunnelling connections are enabled by default. If no incoming serial tunneling is supported by the application program, setting this flag false may improve the robustness of web service by automatically closing a non-web connection so that the application program does not have to.

C: void **Ether_XPort_Defaults** (int modulenum)
4th: **Ether_XPort_Defaults** (modulenum --)

This function works for both the EtherSmart and WiFi Wildcards, restoring the Mosaic default settings to the Lantronix device (XPort or WiPort). It SENDs a message via the ether_command mailbox to the task running the Ether_Service_Loop which dispatches the action function. (If the mailbox is full because the action task hasn't cleared it yet, the SEND routine will PAUSE; see Ether_Ready_For_Command). This function suspends multitasking for several seconds while it enters the monitor mode, resumes multitasking, and sends the default versions of Lantronix flash block0 (serial and network settings), block3 (additional network settings), and block7 (hardware handshaking configuration) s-records. If the specified modulenum belongs to a WiFi Wildcard, records 8 (WiFi settings) and 9 (WiFi encryption key) are also sent to the Lantronix device. This routine then executes the monitor mode RS reset command to instantiate the new values in Lantronix flash memory. The entire operation takes approximately 13 seconds. 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 error flag in the

remaining 2 bytes is placed in the ether_response mailbox; the error flag is also available using Ether_Error. The error flag is zero if the operation was successful, or nonzero if there was a problem that prevented the operation from completing. 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.

Implementation detail: The Lantronix XPort and WiPort documentation manuals are available at www. lantronix.com. Many of the Lantronix default values are retained this function, but a significant number are changed to reflect the Mosaic default values. The Mosaic custom values are as follows: The baud rate is set to 115 Kbaud, hardware handshaking is enabled, the local TCP port is set to 80, the connect configuration is set to "accept all incoming connections, and use modem mode without echo". The flush mode is set such that flushing is disabled, and pack control is enabled. The internal "configuration" HTTP port is set to 8000, and the "+++ passthrough suppression" feature is enabled. The local IP, gateway IP, and netmask are kept at the Lantronix default values of 0.0.0.0 (unassigned), so that the device expect to have these parameters assigned via DHCP (Dynamic Host Configuration Protocol) on the LAN. The hardware handshaking pins are enabled in flash block 7.See also Ether_XPort_Update.

C: void **Ether_XPort_Updat**e (int modulenum)

4th: **Ether_XPort_Update** (modulenum --)

This function works for both the EtherSmart and WiFi Wildcards, restoring the Mosaic default settings to the Lantronix device (XPort or WiPort). For the specified module, it SENDs a message via the ether_command mailbox to the task running the Ether_Service_Loop which dispatches the action function. (If the mailbox is full because the action task hasn't cleared it yet, the SEND routine will PAUSE; see Ether_Ready_For_Command). This function suspends multitasking for several seconds while it enters the monitor mode, resumes multitasking, and sends the versions of Lantronix flash block0 (serial and network settings), block3 (additional network settings), and block7 (hardware handshaking configuration) s-records. The s-records include the default values as optionally modified by the following user-called configuration functions (see their glossary entries for details):

Ether_DHCP_Name
Ether_Gateway
Ether_Internal_Webserver_Port
Ether_Local_IP
Ether_Local_Port
Ether_Telnet_Password (see glossary entry for cautions about use)
Ether_TCP_Control

For a WiFi Wildcard, the following configuration functions are also relevant (see their glossary entries for details):

WiFi_Options
WiFi_SSID
WiFi_Security

This routine then executes the monitor mode RS reset command to instantiate the new values in the Lantronix flash memory. The entire operation takes approximately 13 seconds. 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 error flag in the remaining 2 bytes is placed in the ether_response mailbox; the error flag is also available using Ether_Error. The error flag is zero if the operation was successful, or nonzero if there was a problem that prevented the operation from completing. 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. To revert to the default configuration, see Ether_XPort_Defaults. Note: DHCP (Dynamic Host Configuration Protocol) is enabled by default and after Ether_XPort_Defaults is executed; in this condition, the Lantronix obtains its IP address, gateway IP and netmask via DHCP on the LAN. To disable DHCP, assign a non-zero value to the local IP address, gateway IP, or netmask using the relevant functions listed in this glossary entry followed by Ether_XPort_Update. The presence of the non-zero IP address will override any DHCP value and disable DHCP. Likewise, specifying a non-zero gateway IP address or netmask will override any gateway IP and netmask values and disable DHCP.

C: int **HTTP_Add_GUI_Handler** (xaddr url_xstring, int url_count,
xaddr (*handler_ptr)(void), int modulenum)

4th: **HTTP_Add_GUI_Handler** (url_xaddr\url_count\handler_xcfa \modulenum -- error)

Adds a GUI (Graphical User Interface) handler function with the specified execution address to be called each time the webserver receives a GET statement that references the specified URL (Universal Resource Locator, as described below in this glossary entry). A GUI handler is used to implement a web-based “remote front panel” for an instrument that contains a touchscreen and graphics display. When a user clicks on a screen image (typically the same image that is present on the graphics display) presented in a browser window, the GUI toolkit processes the input as if a touch on the touchscreen had occurred, calling the associated GUI action function, and updating the screen image both on the instrument an on the remote browser screen. A GUI handler is different than a standard webservice handler. A standard non-GUI webservice handler executes automatically without any intervention from the user’s application task, and is dispatched by the Ethernet task running the Ether_Service_Loop routine. A GUI handler, on the other hand, requires an interaction with the GUI Toolkit, and, for task synchronization reasons, must be dispatched from the user’s application task. This HTTP_Add_GUI_Handler function posts the handler function and the associated URL to the autoserve array, and marks the function as a GUI Handler so that it will be dispatched via the application task. This function initializes the row specified by the contents of HTTP_Index_Ptr in the autoserve array. This autoserve array is initialized by Ether_Info_Init or WiFi_Info_Init and their calling functions (see the “Initialization” section of the categorized function list), and it can also be declared using HTTP_Is_Autoserve_Array. Each row associates a URL with the corresponding user-specified HTTP handler function that serves out HTML or image data in response to the specified URL request. Each 8-byte row of the autoserve array contains: a 1byte URL count, a 1-byte URL string page, a 2-byte URL string start addr, and a 4-byte handler xcfa (extended code field address, a function pointer). This routine sets the most significant byte of the handler execution address to mark the handler as a GUI handler that is passed to the application task. This routine writes to the row index specified by the contents of HTTP_Index_Ptr, and then increments the contents of HTTP_Index_Ptr. This function returns zero if there was no error, or a nonzero value if the next available index pointed to by HTTP_Index_Ptr is greater than or equal to the maximum number of rows set by the HTTP_Is_Autoserve_Array function.

Notes about the handler function: Each web handler function posted by this routine must accept a single integer input parameter that specifies the EtherSmart/WiFi modulenum, and must not return any values. Every handler must send a valid HTTP header and content type, typically by executing HTTP_Put_Header and HTTP_Put_Content_Type (see their glossary entries) before sending the referenced buffer. GUI handler functions that implement a web “remote front panel” must perform only one send operation to serve the webpage, using one of the functions HTTP_GUI_Send_Buffer, HTTP_GUI_Send_LBuffer, or HTTP_GUI_Send_2Buffers. The most

common function used to serve out the GUI webpage is HTTP_GUI_Send_2Buffers, with the first buffer containing the HTTP header and imagemap HTML text, and the second buffer containing the bitmap screen image.

Notes about the URL: URL means Universal Resource Locator. In the context of this driver code, it means the string that appears in the web browser's address bar starting with the / character after the domain name or IP address. If only the IP address or corresponding name is present in the address bar, the browser will send the URL as a single / character. Consequently, it is recommended that when defining a set of webserver pages, you always include a URL with a single / character as a synonym for the home page URL, in case the user types only the IP address or machine name. Typically, you'll want to also declare the home page URL as

/index.html

The URL starts with a / character and ends with the character before either a terminating space (ascii 0x20) or a ? (ascii 0x3F) character. If the ? character is present, it indicates the presence of a query field following the ? that typically results from a "form object" or "imagemap" in the HTML code being displayed in the browser window. See the glossary entry for HTTP_Add_Handler for a standard form example. Here we present an "imagemap" example that is used to implement a typical GUI handler web page. See the demo code associated with this driver for a coded example. When an image object is declared as an HTML imagemap, then any click on the image causes the browser to issue a GET command to a specified response URL, passing the X and Y coordinates of the mouse click as a comma delimited query field. The X and Y coordinates are in units of pixels relative to the upper left corner of the image.

Imagemap example: Here is an example of a browser GET request when a mouse click occurs 73 pixels to the right and 103 pixels down from the upper left corner of a map image:

GET /gui_response.html?73,103

The GET is always capitalized and followed by a space, after which the URL appears in the same case as it was typed in the browser's address bar or in the HTML code that invoked it. URL's are case sensitive. To properly manage HTTP GUI handlers, the main user application task (the same task that invokes the GUI functions) must call Ether_Check_GUI periodically to accept any incoming GUI web requests and dispatch the associated handler that was posted by this HTTP_Add_GUI_Handler function. In the HTTP GUI handler function associated with the URL string

/gui_response.html

the HTTP_Imagemap function (see its glossary entry) should be called to extract and return the X and Y coordinates from the query field. Then the Simulated_Touch_To_Image function (defined in the special GUI Toolkit section of this glossary document) can be invoked to simulate the touch at the specified screen coordinates, draw the press and release graphics, activate the screen button's press handler, and, if the screen image changed, reload the screen image buffer with the updated screen bitmap image. Then HTTP_GUI_Send_2Buffers can be used to send the HTTP header and HTML text (in the first buffer) and graphics image (in the second buffer) to the browser to complete the operation. Careful examination of the demo code should clarify this technique.

C: int **HTTP_Add_Handler** (xaddr url_xstring, int url_count, xaddr (*handler_ptr)(void),
int modulenum)

4th: **HTTP_Add_Handler** (url_xaddr\url_count\handler_xcfa\modulenum -- error)

Adds a handler function with the specified execution address to be called each time the webserver receives a GET statement that references the specified URL (Universal Resource

Locator, as described below in this glossary entry). Initializes the row specified by the contents of HTTP_Index_Ptr in the autoserve array. This autoserve array is initialized by Ether_Info_Init or WiFi_Info_Init and their calling functions (see the "Initialization" section of the categorized function list), and it can also be declared using HTTP_Is_Autoserve_Array. Each row associates a URL with the corresponding user-specified HTTP handler function that serves out HTML or image data in response to the specified URL request. Each 8-byte row of the autoserve array contains: a 1byte URL count, a 1-byte URL string page, a 2-byte URL string start addr, and a 4-byte handler xcfa (extended code field address, a function pointer). This routine clears the most significant byte of the handler execution address to mark the handler as a standard handler that is executed by the ethertask (as opposed to a GUI handler; see HTTP_Add_GUI_Handler). This function writes to the row specified by the contents of HTTP_Index_Ptr, and then increments the contents of HTTP_Index_Ptr. This function returns zero if there was no error, or a nonzero value if the next available index pointed to by HTTP_Index_Ptr is greater than or equal to the maximum number of rows set by HTTP_Is_Autoserve_Array, or by one of the functions listed in the "Initialization" section of the categorized function list.

Notes about the handler function: Each web handler function posted by this routine must accept a single integer input parameter that specifies the EtherSmart/WiFi modulenum, and must not return any values. Every handler must send a valid HTTP header and content type, typically by executing HTTP_Put_Header and HTTP_Put_Content_Type (see their glossary entries) before sending the referenced buffer. Standard (non-GUI) handler routines can perform as many send operations as needed to serve the page using HTTP_Send_Buffer, HTTP_Send_LBuffer, or HTTP_Send_2Buffers

Notes about the URL: URL means Universal Resource Locator. In the context of this driver code, it means the string that appears in the web browser's address bar starting with the / character after the domain name or IP address. If only the IP address or corresponding name is present in the address bar, the browser will send the URL as a single / character. Consequently, it is recommended that when defining a set of webserver pages, you always include a URL with a single / character as as a synonym for the home page URL, in case the user types only the IP address or machine name. Typically, you'll want to also declare the home page URL as

```
/index.html
```

The URL starts with a / character and ends with the character before either a terminating space (ascii 0x20) or a ? (ascii 0x3F) character. If the ? character is present, it indicates the presence of a query field following the ? that typically results from a "form object" or "imagemap" in the HTML code being displayed in the browser window.

URL Examples: If the Lantronix device is assigned IP address 10.0.1.22, and the user types in the address bar of the browser:

```
10.0.1.22/form_entry.html
```

then the browser opens a connection to port 80 (which we have defined as the EtherSmart/WiFi Wildcard's local port) and sends the following command:

```
GET  /form_entry.html
```

The GET is always capitalized and followed by a space, after which the URL appears in exactly the same case as it was typed in the browser's address bar or in the HTML code that invoked it. URL's are case sensitive. Let's assume that the handler for the URL string /form_entry.html is the one described in the demo code that accompanies this driver. Then when the "submit" button is pressed after the user fills in the form, the browser opens a connection and sends a request of the form:

```
GET  /form_response.cgi?classification=man&name_id=tommy&color=blue
```

In this case the URL is defined as

/form_response.cgi

This is the string that is passed to HTTP_Add_Handler to associate the handler function with the URL. The text after the URL starting with the ? character is the query portion of the URL. Each "field" in the query comprises a fieldname followed by an = character followed by a field value. Fields are separated by the & character. A set of functions including HTTP_Parse_URL, HTTP_Fieldname_Ptr, HTTP_Fieldname_Count, HTTP_Value_Ptr, HTTP_Value_Count, and HTTP_To_Next_Field are available to simplify handling of query fields. Additional functions such as HTTP_Plus_To_Space and HTTP_Unescape deal with the "escaping" of special characters performed by a browser; see their glossary entries for details.

C: **HTTP_AUTOSERVE_DEFAULT_ROWS**
4th: **HTTP_AUTOSERVE_DEFAULT_ROWS**

A 16-bit constant that returns the value 32. This constant is used by Ether_Setup and WiFi_Setup to specify the number of rows in the autoserve array. This constant sets the maximum number of web URLs (Universal Resource Locator strings) that can be responded to by the dynamic webserver. To change the size and/or base xaddress of the autoserve array, use HTTP_Is_Autoserve_Array. See also HTTP_Autoserve_Ptr.

C: xaddr **HTTP_Autoserve_Ptr** (int modulenum)
4th: **HTTP_ Autoserve_Ptr** (modulenum -- xaddr)

Returns the 32-bit extended base address of the autoserve array that is used by the webserver. The base xaddress is set by Is_Autoserve_Array which in turn is called by Ether_Init and WiFi_Init and their calling functions as listed in the "Initialization" section of the categorized function list. Each row in the array associates a URL (Universal Resource Locator, a web address string) with the corresponding user-specified HTTP handler function that serves out HTML or image data in response to the specified URL request. Each 8-byte row of the autoserve array contains: a 1byte URL count, a 1-byte URL string page, a 2-byte URL string start addr, and a 4-byte handler xcfa (extended code field address, a function pointer). HTTP_Is_Autoserve_Array does not erase the declared array; this is done by Ether_Init, WiFi_Init, or their calling functions. If you want to use a pre-initialized flash-based autoserve array, first call Ether_Init, Ether_Setup, WiFi_Init, or WiFi_Setup to create a RAM-based instance of the autoserve array, and then invoke this function to change to a pre-initialized flash version of the array. To add a handler to the autoserve array, see HTTP_Add_Handler and HTTP_Add_GUI_Handler. See the glossary entries for HTTP_Parse_URL and HTTP_Add_Handler for further information about URLs.

C: **HTTP_BINARY_DATA_CONTENT**
4th: **HTTP_BINARY_DATA_CONTENT**

A 16-bit constant that returns the value 5. This constant is passed as a content_identifier to HTTP_Put_Content_Type to indicate that the HTTP header should declare the content type as "application/octet-stream" (raw binary data bytes).

C: void **HTTP_Default_Handler** (int modulenum)
4th: **HTTP_Default_Handler** (modulenum --)

Directly sends a response to the current open Ethernet connection a "404: Not Found" HTTP error response page. The response goes directly from string text stored in flash to the Lantronix; no RAM buffer is used to send the data. This function is automatically called by HTTP_Server when a URL (Universal Resource Locator, or web address string) is received that

does not match a string posted by HTTP_Add_Handler or HTTP_Add_GUI_Handler. You can customize this default handler by defining a function that serves out the appropriate HTTP header followed by a single blank line followed by the desired custom HTML text, and then storing the 32-bit execution address (function pointer) in the HTTP_Default_Handler_Ptr (see its glossary entry). The page served out reminds the user that each URL should start with a / (forward slash) character, and that URLs are case sensitive. This routine serves out the following HTTP header and HTML content, and the webserver then closes the connection:

```
HTTP/1.1 404 Not Found
Server: Mosaic Industries Embedded Webserver
Connection: close
Content-Type: text/html

<html><head><title>404 Not Found</title></head>
<body><H3>Requested URL was not found:</H3>
{the requested URL is printed here}
<p>Note: All URLs should start with the / character and are case sensitive.
</body></html>
```

C: xaddr **HTTP_Default_Handler_Ptr** (int modulenum)
4th: **HTTP_Default_Handler_Ptr** (modulenum -- xaddr)

Returns the 32-bit xaddress in the ether_info struct whose contents are the 32-bit xcfa (extended code field address, or execution address) of the default web handler routine that serves out the "Error 404: Page not found" response. This handler is called by the HTTP_Server when an incoming GET request from the browser specifies a URL (web address string) that has not been posted using HTTP_Add_Handler or HTTP_Add_GUI_Handler. The default contents of this pointer are set to point to HTTP_Default_Handler by Ether_Info_Init or WiFi_Info_Init and their calling functions (see the “Initialization” section of the categorized function list). To change the default handler, create a handler function and store its execution address (32-bit function pointer) in the 32-bit xaddress returned by HTTP_Default_Handler_Ptr. See the glossary entry of HTTP_Default_Handler for more details.

C: xaddr **HTTP_Enable_Ptr** (int modulenum)
4th: **HTTP_Enable_Ptr** (modulenum -- xaddr)

Returns the xaddress of a a 16-bit flag within the ether_info struct for the specified EtherSmart/WiFi module. If the flag is true, the Ether_Command_Manager will input the first carriage-return/linefeed-delimited line from the remote into the HTTP_Inbuf and, if a leading GET substring is detected, will service the incoming web connection. If the contents of HTTP_Enable_Ptr equal zero, no characters will be automatically inputted from passive incoming connections and consequently, no attempt will be made to "identify" the connection as http or non-http. If the application does not support web connections, setting this flag false can simplify the handling of incoming "serial tunneling" connections, as the application program is solely in charge of accepting incoming chars, instead of having to look for the first incoming line placed in the HTTP_Inbuf. The default flag state set at initialization time is true (web service is enabled).

C: int **HTTP_Fieldname_Count** (int modulenum)
4th: **HTTP_Fieldname_Count** (modulenum -- count)

Returns the count of the current query field between the ? (first field only) or & (all other fields) delimiter and the = sign of the URL (Universal Resource Locator string) for the specified

EtherSmart/WiFi module. The count does not include the ? or & leading delimitor, nor the = character or blank trailing delimiter. Returns 0 if the query field is not present or the URL is exhausted. This routine is used during the handling of webservice requests to parse the query field of the URL. When the handler associated with the incoming URL is first called, this function returns the fieldname count of the first query field after the ? (if present). To advance to the next field, call HTTP_To_Next_Field. Each field is of the form:

fieldname=value

Within the current query field, use the functions HTTP_Fieldname_Ptr, HTTP_Fieldname_Count, HTTP_Value_Ptr, and HTTP_Value_Count to identify the fieldname and value parameters passed by the browser's GET command string. Let's assume that the handler for the URL string /form_entry.html is the one described in the demo code that accompanies this driver. Then when the "submit" button is pressed after the user fills in the form, the browser opens a connection and sends a request of the form:

GET /form_response.cgi?classification=man&name_id=tommy&color=blue

In this case the URL is defined as

/form_response.cgi

This is the string that is passed to HTTP_Add_Handler to associate the handler function with the URL. The text after the URL starting with the ? character is the query portion of the URL. Each "field" in the query comprises a fieldname followed by an = character followed by a field value. Fields are separated by the & character. When the handler function is first called, executing Http_Fieldname_Ptr returns the xaddress of the 'c' in 'classification' and the HTTP_Fieldname_Count returns 14, the count of 'classification'. HTTP_Value_Ptr returns the xaddress of the 'm' in 'man', and HTTP_Value_Count returns 3, the count of 'man'. A call to HTTP_To_Next_Field advances the current field to after the next & character. Then Http_Fieldname_Ptr returns the xaddress of the 'n' in 'name_id' and the HTTP_Fieldname_Count returns 7, the count of 'name_id'. HTTP_Value_Ptr then returns the xaddress of the 't' in 'tommy', and HTTP_Value_Count returns 5, the count of 'tommy'. An additional call to HTTP_To_Next_Field advances the current field to after the next & character. Then Http_Fieldname_Ptr returns the xaddress of the 'c' in 'color' and the HTTP_Fieldname_Count returns 5, the count of 'color'. HTTP_Value_Ptr then returns the xaddress of the 'b' in 'blue', and HTTP_Value_Count returns 4, the count of 'blue'. Because the URL is now exhausted, an additional call to HTTP_To_Next_Field would result in HTTP_Fieldname_Count and HTTP_Value_Count returning 0, signalling that there are no additional query fields to be examined. See the glossary entry of HTTP_Imagemap function for information on how to handle an HTTP GUI "remote front panel" web request.

C: xaddr **HTTP_Fieldname_Ptr** (int modulenum)

4th: **HTTP_Fieldname_Ptr** (modulenum -- xaddr)

Returns the 32-bit extended address of the first character in the current query field of the URL (Universal Resource Locator string) for the specified EtherSmart/WiFi module. This routine is used during the handling of webservice requests to parse the query field of the URL. The returned xaddress points to the character after the ? (first field only) or & (all subsequent fields). When the handler associated with the incoming URL is first called, this function points to the first query field after the ? (if present). To advance to the next field, call HTTP_To_Next_Field. Each field is of the form:

fieldname=value

Within the current query field, use the functions HTTP_Fieldname_Ptr, HTTP_Fieldname_Count, HTTP_Value_Ptr, and HTTP_Value_Count to identify the fieldname and value parameters passed by the browser's GET command string. Let's assume that the

handler for the URL string /form_entry.html is the one described in the demo code that accompanies this driver. Then when the "submit" button is pressed after the user fills in the form, the browser opens a connection and sends a request of the form:

GET /form_response.cgi?classification=man&name_id=tommy&color=blue

In this case the URL is defined as

/form_response.cgi

This is the string that is passed to HTTP_Add_Handler to associate the handler function with the URL. The text after the URL starting with the ? character is the query portion of the URL. Each "field" in the query comprises a fieldname followed by an = character followed by a field value. Fields are separated by the & character. When the handler function is first called, executing Http_Fieldname_Ptr returns the xaddress of the 'c' in 'classification' and the HTTP_Fieldname_Count returns 14, the count of 'classification'. HTTP_Value_Ptr returns the xaddress of the 'm' in 'man', and HTTP_Value_Count returns 3, the count of 'man'. A call to HTTP_To_Next_Field advances the current field to after the next & character. Then Http_Fieldname_Ptr returns the xaddress of the 'n' in 'name_id' and the HTTP_Fieldname_Count returns 7, the count of 'name_id'. HTTP_Value_Ptr then returns the xaddress of the 't' in 'tommy', and HTTP_Value_Count returns 5, the count of 'tommy'. An additional call to HTTP_To_Next_Field advances the current field to after the next & character. Then Http_Fieldname_Ptr returns the xaddress of the 'c' in 'color' and the HTTP_Fieldname_Count returns 5, the count of 'color'. HTTP_Value_Ptr then returns the xaddress of the 'b' in 'blue', and HTTP_Value_Count returns 4, the count of 'blue'. Because the URL is now exhausted, an additional call to HTTP_To_Next_Field would result in HTTP_Fieldname_Count and HTTP_Value_Count returning 0, signalling that there are no additional query fields to be examined. See the glossary entry of HTTP_Imagemap function for information on how to handle an HTTP GUI "remote front panel" web request.

C: xaddr **HTTP_Get_Timeout_Msec_Ptr** (int modulenum)

4th: **HTTP_Get_Timeout_Msec_Ptr** (modulenum -- xaddr)

Returns the xaddress within the ether_info struct for the specified EtherSmart/WiFi module that holds a 16-bit timeout used by the Ether_Connection_Manager during the attempt to identify an incoming web connection. If a carriage-return/linefeed-delimited line is not available from the remote 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 of 5000 (5 seconds) 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 delay parameter. See also HTTP_Timeout_Msec_Ptr.

C: void **HTTP_GUI_Send_2Buffers** (xaddr xbuffer1, uint count1, xaddr xbuffer2, uint count2, int modulenum)

4th: **HTTP_GUI_Send_2Buffers** (xbuffer1\count1\xbuffer2\count2\module--)

This function serves out web content to implement a web-based "remote front panel" for GUI (Graphical User Interface) touchscreen instruments. This function is coded into the user-specified handler function that is posted by HTTP_Add_GUI_Handler; the handler is invoked at runtime by Ether_Check_GUI running in the main application task when the webserver detects a URL that was declared as corresponding to a GUI handler. HTTP_GUI_Send_2Buffers is typically called in the user-specified handler after HTTP_Imagemap extracts the X,Y coordinates

of the mouse click on the web-based screen image, and after Simulated_Touch_To_Image (defined in the GUI Toolkit driver) invokes the GUI routines and refreshes the screen image in the image buffer. HTTP_GUI_Send_2Buffers is used to send the HTTP header and HTML text in one buffer (see HTTP_Put_Header and HTTP_Put_Content_Type), followed immediately by sending the image data in another buffer, all combined as a single command. Http_GUI_Send_2Buffers stores the specified input parameters into the ether_info struct and SENDs a message via the ether_command mailbox to the task running the Ether_Service_Loop which dispatches the action function. (If the mailbox is full because the action task hasn't cleared it yet, the SEND routine will PAUSE; see Ether_Ready_For_Command). Sends to the EtherSmart/WiFi Wildcard up to count1 bytes of data starting at the 32-bit extended memory address xbuffer1, and then sends up to count2 bytes of data starting at the 32-bit extended memory address xbuffer2. Returns the total number of bytes sent. The total number of bytes actually sent is available via Ether_Numbytes_Sent. Does not send bytes if there is no connection, or if there has been a change in connection status (e.g., a transient disconnect) as detected by Ether_Disconnect_During_Send; in these cases, this routine increments the connection status value returned by Ether_Connect_Status to an odd value to flag the transient disconnect event. After the transmission of the two buffers has completed, the web connection will be automatically closed by the Ethernet task. Any required end of line characters such as carriage return (0x0D) and linefeed (0x0A) characters must be in the buffer; they are not added by this routine. Each of the two send operations exits within the time specified by the contents of HTTP_Timeout_Msec_Ptr whether or not the maximum number of bytes have been sent.
NOTE: Unlike Send_Buffer, this routine does not SEND the numbytes_sent return result via the ether_response mailbox back to the application program.
NOTE: A special rule applies to GUI-based web service routines because of task synchrony considerations: Each web response must be completed using a single send command. If an image is to be served out, then use this HTTP_GUI_Send_2Buffers function to send the HTTP header and required HTML tag text in xbuffer1, plus the separate image in xbuffer2.
Usage Notes: Note that a standard non-GUI webservice handler executes automatically without any intervention from the user's application task, and is dispatched by the Ethernet task running the Ether_Service_Loop routine. On the other hand, a GUI handler is different than a standard webservice handler in that it requires an interaction with the GUI Toolkit, and, for task synchronization reasons, must be dispatched from the user's application task. A GUI handler is used to implement a web-based "remote front panel" for an instrument that contains a touchscreen and graphics display. HTTP_Add_GUI_Handler posts the handler (the function containing the call to HTTP_GUI_Send_2Buffers) along with the associated URL to the autoserve array, and marks the function as a GUI Handler so that it will be dispatched via the application task from Ether_Check_GUI. When a user clicks on a screen image (typically the same image that is present on the graphics display) presented in a browser window and declared using the "imagemap" HTML attribute, the browser returns the X,Y coordinates in a query field. These coordinates are extracted by the HTTP_Imagemap function; see its glossary entry. By invoking the GUI toolkit function named Simulated_Touch_To_Image (see the GUI toolkit documentation), the GUI toolkit processes the input as if a touch on the touchscreen had occurred, calling the associated GUI action function, and updating the screen image both on the instrument and in a specified image buffer. Then this HTTP_GUI_Send_2Buffers function is used to serve the HTTP header and HTML text in xbuffer1, and the bitmap screen image data in xbuffer2 to the remote browser screen. The text xbuffer must include exactly 1 blank line between the HTTP header and and the HTML text; see HTTP_Put_Header and HTTP_Put_Content_Type.

C: void **HTTP_GUI_Send_Buffer** (xaddr xbuffer, uint count, int modulenum)
4th: **HTTP_GUI_Send_Buffer** (xbuffer\count\modulenum --)

This function serves out web content to implement a web-based "remote front panel" for GUI (Graphical User Interface) touchscreen instruments. This function is coded into the user-specified handler function that is posted by HTTP_Add_GUI_Handler; the handler is invoked at runtime by Ether_Check_GUI which must be called periodically by the main application task. Http_GUI_Send_Buffer stores the specified input parameters into the ether_info struct for the specified modulenum and SENDs a message via the ether_command mailbox to the task running the Ether_Service_Loop which dispatches the action function. (If the mailbox is full because the action task hasn't cleared it yet, the SEND routine will PAUSE; see Ether_Ready_For_Command). Sends to the EtherSmart/WiFi Wildcard up to count bytes of data starting at the 32-bit extended memory address xbuffer. The total number of bytes actually sent is available via Ether_Numbytes_Sent. Does not send bytes if there is no connection, or if there has been a change in connection status (e.g., a transient disconnect) as detected by Ether_Disconnect_During_Send; in these cases, this routine increments the connection status value returned by Ether_Connect_Status to an odd value to flag the transient disconnect event. After the transmission, the web connection will be automatically closed by the Ethernet task. Any required end of line characters such as carriage return (0x0D) and linefeed (0x0A) characters must be in the buffer; they are not added by this routine. This function exits within the time specified by the contents of HTTP_Timeout_Msec_Ptr whether or not the maximum number of bytes have been sent.

NOTE: Unlike Send_Buffer, this routine does <u>not</u> SEND the numbytes_sent return result via the ether_response mailbox back to the application program.

NOTE: A special rule applies to GUI-based web service routines because of task synchrony considerations: Each web response <u>must</u> be completed using a <u>single</u> send command. If an image is to be served out, then use the HTTP_GUI_Send_2Buffers function to send the HTTP header and required HTML tag text in one buffer, followed by the separate image buffer in a single send command.

Usage Notes: A GUI handler is used to implement a web-based "remote front panel" for an instrument that contains a touchscreen and graphics display. When a user clicks on a screen image (typically the same image that is present on the graphics display) presented in a browser window, the GUI toolkit processes the input as if a touch on the touchscreen had occurred, calling the associated GUI action function, and updating the screen image both on the instrument an on the remote browser screen. A standard non-GUI webservice handler executes automatically without any intervention from the user's application task, and is dispatched by the Ethernet task running the Ether_Service_Loop routine. A GUI handler is different than a standard webservice handler in that it requires an interaction with the GUI Toolkit, and, for task synchronization reasons, must be dispatched from the user's application task. The HTTP_Add_GUI_Handler function posts the handler function and the associated URL to the autoserve array, and marks the function as a GUI Handler so that it will be dispatched via the application task from Ether_Check_GUI. See also HTTP_Send_2Buffers, HTTP_Put_Header, and HTTP_Put_Content_Type.

C: void **HTTP_GUI_Send_LBuffer** (xaddr xlbuffer, int modulenum)
4th: **HTTP_GUI_Send_LBuffer** (xlbuffer\modulenum --)

Sends the data contents of a "long buffer" which contains a 16-bit data byte count followed by the data bytes. This function "unpacks" the specified long buffer by fetching the 16-bit count from xlbuffer, and passing xlbuffer+2 and the extracted count to HTTP_GUI_Send_Buffer. See the glossary entry for HTTP_GUI_Send_Buffer.

C: **HTTP_IMAGE_BITMAP_CONTENT**
4th: **HTTP_IMAGE_BITMAP_CONTENT**

A 16-bit constant that returns the value 1. This constant is passed as a content_identifier to HTTP_Put_Content_Type to indicate that the HTTP header should declare the content type as "image/bmp" (a bitmap image). It can also be passed as a format_id to the GUI functions Screen_To_Image and Graphic_To_Image (see the special GUI Toolkit section of this glossary document).

C: **HTTP_IMAGE_GIF_CONTENT**
4th: **HTTP_IMAGE_GIF_CONTENT**

A 16-bit constant that returns the value 3. This constant is passed as a content_identifier to HTTP_Put_Content_Type to indicate that the HTTP header should declare the content type as "image/gif" (an image compressed using the GIF format).

C: **HTTP_IMAGE_JPEG_CONTENT**
4th: **HTTP_IMAGE_JPEG_CONTENT**

A 16-bit constant that returns the value 4. This constant is passed as a content_identifier to HTTP_Put_Content_Type to indicate that the HTTP header should declare the content type as "image/jpeg" (an image compressed using the JPEG format).

C: **HTTP_IMAGE_PNG_CONTENT**
4th: **HTTP_IMAGE_PNG_CONTENT**

A 16-bit constant that returns the value 2. This constant is passed as a content_identifier to HTTP_Put_Content_Type to indicate that the HTTP header should declare the content type as "image/png" (an image compressed using the PNG format).

C: TWO_INTS **HTTP_Imagemap** (int modulenum)
4th: **HTTP_Imagemap** (modulenum -- y\x)

Extracts and returns the X and Y coordinates (relative to the upper left image corner) from the query field of an imagemap GET request from a web browser. Returns X = Y = -1 if the required comma is not found in the query field between X and Y. Returns garbage if the query field does not contain valid decimal integers before and after the required comma. Numeric conversion is performed in decimal base. The Forth version of the function returns Y under X. Because a C function can return only a single value, the X and Y values returned by the C function are considered to be a single 32-bit long that is typecast as a union equivalent to two 16-bit integers, with X in the most significant (ms) integer, and Y in the least significant (ls) integer. The types.h header file includes the following typedef that specifies the C return parameter:

```
typedef union
{  ulong int32;
struct
{  int   msInt;
int   lsInt;
} twoNums;
}   TWO_INTS;
```

When an image object is declared as an HTML imagemap, then any click on the image causes the browser to issue a GET command to a specified response URL, passing the X and Y coordinates of the mouse click as a comma delimited query field. The X and Y coordinates are

in units of pixels relative to the upper left corner of the image. An imagemap is declared as follows:

```
<a href="/ gui_response.html">
<img src="/mosaic_logo.gif" ismap></a>
```

When a mouse click occurs, say, 73 pixels to the right and 103 pixels down from the upper left corner of a map image:

```
GET /gui_response.html?73,103
```

The GET is always capitalized and followed by a space, after which the URL appears in exactly the same case as it was typed in the browser's address bar or in the HTML code that invoked it. URL's are case sensitive. To properly manage HTTP GUI handlers, the main user application task (the same task that invokes the GUI functions) must call Ether_Check_GUI periodically to accept any incoming GUI web requests and dispatch the associated handler that was posted by this HTTP_Add_GUI_Handler function. In the HTTP GUI handler function associated with the URL string

```
/gui_response.html
```

this HTTP_Imagemap function is called to extract and return the X and Y coordinates from the query field. Then the Simulated_Touch_To_Image function can be invoked to simulate the touch at the specified screen coordinates, draw the press and release graphics, activate the screen button's press handler, and, if the screen image changed, reload the screen image buffer with the updated screen bitmap image. Then HTTP_GUI_Send_2Buffers can be used to send the HTTP header and HTML text (in the first buffer) and graphics image (in the second buffer) to the browser to complete the operation. Careful examination of the demo code should clarify this technique.

C: xaddr **HTTP_Inbuf** (int modulenum)
4th: **HTTP_Inbuf** (modulenum -- xaddr)

Returns the 32-bit extended base address of the default HTTP web service buffer for incoming data from the TCP/IP link for the specified module. When a passive incoming connection is detected by the Ether_Connection_Manager running in the Ethernet task, it puts the first carriage-return/linefeed-delimited line into HTTP_Inbuf as a counted lstring (up to a maximum of HTTP_Inbufsize bytes, waiting up to HTTP_Get_Timeout_Msec) and looks for a leading GET substring to decide whether the connection is HTTP or serial tunneling. If it is HTTP, the web server is called to parse the URL (web address) in place in the HTTP_Inbuf and the web service request is automatically fulfilled by dispatching the posted handler for the URL. See the glossary entries for HTTP_Parse_URL and HTTP_Add_Handler for further information about URLs. Note that if the connection is not identified as HTTP (because a leading GET substring is not found), the incoming bytes accepted by the Ether_Connection_Manager remain in the HTTP_Inbuf as an lstring (16-bit count followed by the data bytes) where they can be processed by the application program. In this case, the application program learns of the accepted passive connection by polling the Ether_Connect_Status and/or Ether_Passive_Non_Web_Connection on each pass through the program loop routine; see their glossary entries. See also the glossary entries for HTTP_Inbufsize and HTTP_Set_Inbuf.

C: uint **HTTP_Inbufsize** (int modulenum)
4th: **HTTP_Inbufsize** (modulenum -- u)

Returns the size of the HTTP_Inbuf default input buffer for the specified EtherSmart/WiFi module which is set by HTTP_Set_Inbuf and/or by Ether_Init, WiFi_Init and their calling functions as listed in the "Initialization" section of the categorized function list. Note that the allocated buffer size must be 2 bytes bigger than the maxnumbytes parameter passed to

Ether_Init, WiFi_Init or HTTP_Set_Inbuf; these 2 bytes provide room to store the 16- bit count at the start of the Ibuffer. The Ether_Setup and WiFi_Setup functions initialize HTTP_Inbufsize to HTTP_INBUFSIZE_DEFAULT = 254, with an allocated buffer size of 256 bytes. See HTTP_Inbuf.

C: **HTTP_INBUFSIZE_DEFAULT**
4th: **HTTP_INBUFSIZE_DEFAULT**
A 16-bit constant that returns the value 254. This constant is used by Ether_Setup and WiFi_Setup to specify the size of the HTTP_Inbuf buffer and is returned by HTTP_Inbufsize after Ether_Setup or WiFi_Setup is executed. The buffer is allocated as 256 bytes, comprising a 2-byte count stored in the first 2 bytes of the buffer, followed by the 254 maximum bytes of data in the buffer.

C: xaddr **HTTP_Index_Ptr** (int modulenum)
4th: **HTTP_Index_Ptr** (modulenum -- xaddr)
Returns the xaddress within the ether_info struct for the specified EtherSmart/WiFi module that holds a 16-bit index to the next available row in the HTTP autoserve array. The value of this index is used by HTTP_Add_Handler and HTTP_Add_GUI_Handler to specify the row in the autoserve array that will receive the next handler/URL pair. The contents of HTTP_Index_Ptr are zeroed by Ether_Info_Init, WiFi_Info_Init and their callers (see the “Initialization” section of the categorized function list) and incremented by HTTP_Add_Handler and HTTP_Add_GUI_Handler. This index typically does not need to be modified by the programmer.

C: void **HTTP_Is_Autoserve_Array** (xaddr xarraybase, int numrows, int modulenum)
4th: **HTTP_Is_Autoserve_Array** (xarray_xbase\numrows\modulenum --)
Specifies the 32-bit extended base xarraybase and number of rows in the HTTP autoserve array that is used by the webserver. Each row associates a URL (Universal Resource Locator, a web address string) with the corresponding user-specified HTTP handler function that serves out HTML or image data in response to the specified URL request. Each 8-byte row of the autoserve array contains: a 1byte URL count, a 1-byte URL string page, a 2-byte URL string start addr, and a 4-byte handler xcfa (extended code field address, a function pointer). HTTP_Is_Autoserve_Array does not erase the declared array; this is done by Ether_Init or WiFi_Init and theif calling functions as listed in the “Initialization” section of the categorized function list. If you want to use a flash-based autoserve array, first call one of these initialization functions to create a RAM-based instance of the autoserve array, and then invoke this HTTP_Is_Autoserver_Array function to change to a pre-initialized flash version of the array. To add a handler to the autoserve array, see HTTP_Add_Handler and HTTP_Add_GUI_Handler. See the glossary entries for HTTP_Parse_URL and HTTP_Add_Handler for further information about URLs. See also HTTP_AUTOSERVE_DEFAULT_ROWS and HTTP_Autoserve_Ptr.

C: uint **HTTP_Numbytes_Sent** (int modulenum)
4th: **HTTP_Numbytes_Sent** (modulenum -- u)
Returns the contents of the 16-bit numbytes_sent field located in the ether_info structure for the specified EtherSmart/WiFi module. This variable is set by the functions that send data, including Ether_Send_Buffer, Ether_Send_LBuffer, Ether_Send_2Buffers, HTTP_Send_Buffer, HTTP_Send_LBuffer, HTTP_Send_2Buffers, HTTP_GUI_Send_Buffer, HTTP_ GUI_Send_LBuffer, and HTTP_ GUI_Send_2Buffers. The first 3 functions mentioned also return the number of bytes sent as the least significant 16-bits in ether_response.

C: xaddr **HTTP_Outbuf** (int modulenum)
4th: **HTTP_Outbuf** (modulenum -- xaddr)

Returns the 32-bit extended base address of the default web service buffer for outgoing bytes to the TCP/IP link for the given module. This buffer is available for use by the programmer to code web service handlers, and is typically (but not necessarily) passed as the target output buffer to functions such as HTTP_Put_Header, HTTP_Put_Content_Type, HTTP_Send_Buffer, HTTP_Send_LBuffer, HTTP_Send_2Buffers, HTTP_GUI_Send_Buffer, HTTP_GUI_Send_LBuffer, and HTTP_GUI_Send_2Buffers. Note that HTTP_Send_LBuffer and HTTP_GUI_Send_LBuffer expect to find the 16-bit count stored in the first 2 bytes of the buffer, with the data following. The convenient HTTP_Outbuf_Cat function adds a specified string to HTTP_Outbuf. See also HTTP_Set_Outbuf.

NOTE: HTTP_Outbuf or another dedicated buffer should be used by HTTP handler functions to hold web data content. If services other than HTTP are in use, general purpose buffers such as Ether_Outbuf should not be used for HTTP web service handlers which are dispatched from the Ethernet task, as the asynchronous user and Ethernet tasks might try to write conflicting data into a buffer that is not dedicated to a single service.

C: void **HTTP_Outbuf_Cat** (xstring\count\modulenum --)
4th: **HTTP_Outbuf_Cat** (xaddr xstring, uint count, int modulenum)

Concatenates the specified string plus a terminating carriage return and linefeed to the specified module's HTTP_Outbuf buffer, and increments the buffer count that is stored in the first two bytes of HTTP_Outbuf. The string is specified by the extended 32-bit address xstring of its first character, and has count bytes to be added to the HTTP_Outbuf buffer. An end-of-line sequence equal to 0x0D0A (carriage return and linefeed) is post-pended to the HTTP_Outbuf after the string is added. This routine clamps the maximum number of bytes in the destination buffer to the value returned by HTTP_Outbufsize (this value does not include the 2-byte count stored at the start of Ether_Outbuf). This function is useful for building up strings to be sent out via web handler functions via the dynamic webserver.

NOTE: Before adding the first string, be sure to store a 16-bit zero into the xaddress returned by HTTP_Outbuf for the specified module; this initializes the counted lbuffer to its starting size of zero.

C: uint **HTTP_Outbufsize** (int modulenum)
4th: **HTTP_Outbufsize** (modulenum -- u)

Returns the size of the HTTP_Outbuf default web service output buffer for the specified EtherSmart/WiFi module which is set by HTTP_Set_Outbuf and/or Ether_Init, WiFi_Init and their calling functions as listed in the "Initialization" section of the categorized word list. This buffer is available for use by the programmer to code web service handlers. Note that the allocated buffer size must be 2 bytes bigger than the maxnumbytes parameter passed to the initializing function; these 2 bytes provide room to store the 16- bit count at the start of the lbuffer. The Ether_Setup and WiFi_Setup functions initialize HTTP_Outbufsize to HTTP_OUTBUFSIZE_DEFAULT = 1022, with an allocated buffer size of 1024 bytes (1 Kbyte).

C: **HTTP_OUTBUFSIZE_DEFAULT**
4th: **HTTP_OUTBUFSIZE_DEFAULT**

A 16-bit constant that returns the value 1022. This constant is used by Ether_Setup and WiFi_Setup to specify the size of the HTTP_Outbuf buffer and is returned by HTTP_Outbufsize

after initialization. The buffer is allocated as 1024 bytes, comprising a 2-byte count stored in the first 2 bytes of the buffer, followed by the 1022 maximum bytes of data in the buffer.

C: void **HTTP_Parse_URL** (xaddr xstring, int max_count, int modulenum)

4th: **HTTP_Parse_URL** (xaddr xstring\max_count\modulenum --)

Parses the URL (Universal Resource Locator, as described below in this glossary entry) by setting several pointers and string length variables that are used by additional user-callable parsing utilities. This function is not typically invoked by the programmer, because it is automatically called by the Ethernet task running Ether_Service_Loop when a passive incoming connection is identified as a web connection by the presence of a leading GET substring. The input parameter xstring specifies the 32-bit xaddress of the first character in the string which starts after the space after GET in the HTTP_Inbuf buffer. The max_count parameter is constrained by HTTP_Inbufsize, and may include more characters and/or more lines than the URL itself. This routine skips any leading spaces (typically not present), then scans for the required trailing space after the URL, then sets HTTP_URL_Full_Count, scans for the ? char (if any) which starts the query portion of the url and stores its offset (or the offset to the final BL) in HTTP_Fieldname_Ptr, then un-escapes the base portion (not including query fields) of the resulting URL, and sets HTTP_URL_Base_Count. The unescape process replaces each HTTP 3-character escape sequence %hexhex (where hex is a hexadecimal digit) with its equivalent single ascii character corresponding to the 2-digit hexadecimal value. If there was no trailing blank in the url buffer as required by the HTTP standard, then this routine sets HTTP_URL_Full_Count = HTTP_URL_Base_Count = 0. This function enables the operation of a set of user-callable functions including HTTP_Parse_URL, HTTP_Fieldname_Ptr, HTTP_Fieldname_Count, HTTP_Value_Ptr, HTTP_Value_Count, HTTP_To_Next_Field, as HTTP_Plus_To_Space, and HTTP_Unescape to simplify the handling of query fields by HTTP handler functions. See the glossary entry of HTTP_Imagemap function for information on how to handle an HTTP GUI “remote front panel” web request.

Notes about the URL: URL means Universal Resource Locator. In the context of this driver code, it means the string that appears in the web browser’s address bar starting with the / character after the domain name or IP address. If only the IP address or corresponding name is present in the address bar, the browser will send the URL as a single / character. Consequently, it is recommended that when defining a set of webserver pages, you always include a URL with a single / character as as a synonym for the home page URL, in case the user types only the IP address or machine name. Typically, you’ll want to also declare the home page URL as

```
/index.html
```

The URL starts with a / character and ends with either a terminating space (ascii 0x20) that is not included in the URL string countl, or with a ? (ascii 0x3F) character. If the ? character is present, it indicates the presence of a query field following the ? that typically results from a “form object” or “imagemap” in the HTML code being displayed in the browser window.

URL Examples: If the Lantronix device is assigned IP address 10.0.1.22, and the user types in the address bar of the browser:

```
10.0.1.22/form_entry.html
```

then the browser opens a connection to port 80 (which we have defined as the EtherSmart/WiFi Wildcard’s local port) and sends the following command:

```
GET  /form_entry.html
```

The GET is always capitalized and followed by a space, after which the URL appears in exactly the same case as it was typed in the browser’s address bar or in the HTML code that invoked it. URL’s are case sensitive. Let’s assume that the handler for the URL string /form_entry.html is

the one described in the demo code that accompanies this driver. Then when the "submit" button is pressed after the user fills in the form, the browser opens a connection and sends a request of the form:

GET /form_response.cgi?classification=man&name_id=tommy&color=blue

In this case the URL is defined as

/form_response.cgi

This is the string that is passed to HTTP_Add_Handler to associate the handler function with the URL. The text after the URL starting with the ? character is the query portion of the URL. Each "field" in the query comprises a fieldname followed by an = character followed by a field value. Fields are separated by the & character. A set of functions including HTTP_Parse_URL, HTTP_Fieldname_Ptr, HTTP_Fieldname_Count, HTTP_Value_Ptr, HTTP_Value_Count, and HTTP_To_Next_Field are available to simplify handling of query fields. Additional functions such as HTTP_Plus_To_Space and HTTP_Unescape deal with the "escaping" of special characters performed by a browser; see their glossary entries for details.

C: void **HTTP_Plus_To_Space** (xaddr xstring, int count)
4th: **HTTP_Plus_To_Space** (xstring\cnt --)

This function is handy for processing forms data sent by a browser as a GET statement. The browser typically replaces each space in a value string of a query field with a + sign. This function accepts a substring in RAM whose first character is at the 32-bit xaddress xstring, with count characters. This substring is typically generated by the functions HTTP_Value_Ptr and HTTP_Value_Count (see their glossary entries). This function replaces any embedded ascii + characters in the specified substring with an ascii blank.

C: void **HTTP_Put_Content_Type** (xaddr xlbuffer, uint max_bufsize, int dynamic, int content_id)
4th: **HTTP_Put_Content_Type** (xlbuffer\max_bufsize\dynamic\content_id --)

Appends to the specified counted xlbuffer a substring containing the content type, an optional header line that prevents cacheing by the browser for dynamic webpages, and a single blank line to end the HTTP header. The xlbuffer is typically HTTP_Outbuf (see its glossary entry), and the max_bufsize is typically HTTP_Outbufsize, but other buffers dedicated to HTTP use can be specified. Adds the number of placed characters to the string count stored in the 2 bytes at xlbuffer, and appends the ascii characters following the initial buffer contents (as typically placed by HTTP_Put_Header). The content_id is one of the following:

HTTP_TEXT_HTML_CONTENT	HTTP_TEXT_PLAIN_CONTENT
HTTP_IMAGE_BITMAP_CONTENT	HTTP_IMAGE_PNG_CONTENT
HTTP_IMAGE_GIF_CONTENT	HTTP_IMAGE_JPEG_CONTENT
HTTP_BINARY_DATA_CONTENT	

The names of these constants are self explanatory, but you can consult their glossary entries for more details. This function places the content type string terminated with a carriage return and linefeed (crlf), and, if the dynamic input flag is true (nonzero), the string:

Cache-Control: no-cache

with its terminating crlf is appended to the buffer. If the dynamic flag is zero, this Cache-Control line is not appended. In all cases, a final crlf is appended to end the header with the required empty line. See the glossary entry for HTTP_Put_Header for a discussion of the required browser configuration for the EtherSmart/WiFi embedded webserver when mixed content (other than all text) is served out in a single page.

Notes about the no-cache directive: This is very important for dynamic web pages. If cacheing by the browser is not disabled, then the browser will feel free to use a version of the webpage that was served out via an earlier web request. While this is very efficient for static web page

content, it is a real problem if you want the web page to display the latest dynamic data such as the states of inputs, a system clock time, or the current state of the graphics screen.

C: void **HTTP_Put_Header** (xaddr xlbuffer, uint max_bufsize)
4th: **HTTP_Put_Header** (xlbuffer\max_bufsize --)
Writes the following into the counted long buffer xlbuffer with maximum size max_bufsize:

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 2 bytes at xlbuffer, with the ascii characters following. With the addition of a valid content type, an optional line that prevents cacheing by the browser for dynamic webpages, and a single blank line to end the header as placed by HTTP_Put_Content_Type (see its glossary entry), this provides the required HTTP header required for a valid response to a GET request from a web browser. This HTTP header is not to be confused with the <head> portion of an HTML web page. All valid web responses, whether they include text or images, should begin with a valid HTTP header. This header contains only the essential fields necessary for this simple webserver. The HTTP/1.1 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 (Universal Resource Locator; see HTTP_Add_Handler and HTTP_Add_GUI_Handler). 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 0x0D0A (carriage return/linefeed) MUST follow the header; HTTP_Put_Content_Type does this for you.
Important Browser Notes: The Lantronix hardware on the EtherSmart or 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, text/html and image/bmp) are present in a single webpage. 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. We highly recommend the use of the free Opera web browser available for download at www.opera.com. Simply go to www.opera.com and select "Download Opera". The download and install are quick, and the program is compact. And, it's very easy to configure for the single-connection 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 the Opera web browser with the EtherSmart/WiFi Wildcard dynamic webserver.

C: uint **HTTP_Send_2Buffers** (xaddr xbuffer1, uint count1, xaddr xbuffer2, uint count2,
int modulenum)
4th: **HTTP_Send_2Buffers** (xbuffer1\count1\ xbuferf2\count2\module--numbytes_sent)
Sends to the EtherSmart/WiFi Wildcard up to count1 bytes of data starting at the 32-bit extended memory address xbuffer1, and then sends up to count2 bytes of data starting at the

32-bit extended memory address xbuffer2. Returns the total number of bytes actually sent. Unlike the Send_2Buffers function which is called directly by the main application task, this HTTP_Send_2Buffers function is called from within a user-specified webserver handler function which is invoked by the Ethernet task running the Ether_Service_Loop. (See HTTP_Add_Handler for details about how to post a web handler function). HTTP_Send_2Buffers does not send bytes if there is no connection, or if there has been a change in connection status (e.g., a transient disconnect) as detected by Ether_Disconnect_During_Send; in these cases, this routine increments the connection status value returned by Ether_Connect_Status to an odd value to flag the transient disconnect event. Any required end of line characters such as carriage return (0x0D) and linefeed (0x0A) characters must be in the buffers; they are not added by this routine. Each of the two send operations exits within within the time specified by the contents of HTTP_Timeout_Msec_Ptr whether or not the maximum number of bytes have been sent. See also HTTP_GUI_Send_2Buffers, HTTP_Put_Header, and HTTP_Put_Content_Type.

C: uint **HTTP_Send_Buffer** (xaddr xbuffer, uint count, int modulenum)
4th: **HTTP_Send_Buffer** (xbuffer\count\modulenum -- numbytes_sent)
Sends to the EtherSmart/WiFi Wildcard up to count bytes of data starting at the 32-bit extended memory address xbuffer, and returns the number of bytes actually sent. Unlike the Send_Buffer function which is called directly by the main application task, this HTTP_Send_Buffer function is called from within a user-specified webserver handler function which is invoked by the Ethernet task running the Ether_Service_Loop. (See HTTP_Add_Handler for details about how to post a web handler function). HTTP_Send_Buffer does not send bytes if there is no connection, or if there has been a change in connection status (e.g., a transient disconnect) as detected by Ether_Disconnect_During_Send; in these cases, this routine increments the connection status value returned by Ether_Connect_Status to an odd value to flag the transient disconnect event. Any required end of line characters such as carriage return (0x0D) and linefeed (0x0A) characters must be in the buffer; they are not added by this routine. This function exits within the time specified by the contents of HTTP_Timeout_Msec_Ptr whether or not the maximum number of bytes have been sent. See also HTTP_Send_LBuffer, HTTP_GUI_Send_Buffer, HTTP_Put_Header, and HTTP_Put_Content_Type.

C: uint **HTTP_Send_LBuffer** (xaddr xlbuffer, int modulenum)
4th: **HTTP_Send_LBuffer** (xlbuffer\modulenum -- numbytes_sent)
The xlbuffer parameter is a 32-bit extended address that holds the 16-bit buffer count followed by the buffer data. Fetches the count from xlbuffer and sends to the EtherSmart/WiFi Wildcard up to count bytes of data that are located starting at xlbuffer+2. Returns the number of bytes actually sent. Unlike the Send_LBuffer function which is called directly by the main application task, this HTTP_Send_LBuffer function is called from within a user-specified webserver handler function which is invoked by the Ethernet task running the Ether_Service_Loop. (See HTTP_Add_Handler for details about how to post a web handler function). HTTP_Send_LBuffer does not send bytes if there is no connection, or if there has been a change in connection status (e.g., a transient disconnect) as detected by Ether_Disconnect_During_Send; in these cases, this routine increments the connection status value returned by Ether_Connect_Status to an odd value to flag the transient disconnect event. Any required end of line characters such as carriage return (0x0D) and linefeed (0x0A) characters must be in the buffer; they are not added by this routine. This function exits within the time specified by the contents of HTTP_Timeout_Msec_Ptr whether or not the maximum number of bytes have been sent. See

also HTTP_Send_Buffer, HTTP_GUI_Send_LBuffer, HTTP_Put_Header, and HTTP_Put_Content_Type.

C: uint **HTTP_Server** (int modulenum)
4th: **HTTP_Server** (modulenum --)

This function is automatically called by the Ether_Connection_Manager which is invoked by the Ether_Service_Loop in the Ethernet task. It is a utility function that is typically not invoked by name. This function serves out a web page to the current connection that has been identified as HTTP. It looks up the incoming URL in the autoserve array and, if found, calls the associated handler and then closes the connection. If the URL is not found, HTTP_Default_Handler is called to serve out the "Page not found" error to the browser. Each web handler function posted by HTTP_Add_Handler or HTTP_Add_GUI_Handler must accept a single integer input parameter that specifies the EtherSmart/WiFi modulenum, and must not return any values. Every handler must send a valid HTTP header and content type, typically by executing HTTP_Put_Header and HTTP_Put_Content_Type before sending the referenced page content. Standard (non-GUI) handler routines can perform as many send operations as needed to serve the page using HTTP_Send_Buffer, HTTP_Send_LBuffer, or HTTP_Send_2Buffers. GUI handler functions that implement a web "remote front panel" must perform only one send operation to serve the page, using one of the functions using HTTP_GUI_Send_Buffer, HTTP_GUI_Send_LBuffer, or HTTP_GUI_Send_2Buffers; the latter function is the most commonly used. See the glossary entries of HTTP_Add_Handler or HTTP_Add_GUI_Handler for more information.

Implementation details: When a passive incoming connection is detected by the Ether_Connection_Manager running in the Ethernet task, it puts the first carriage-return/linefeed-delimited line into HTTP_Inbuf as a counted lstring (up to a maximum of HTTP_Inbufsize bytes) and looks for a leading GET substring to decide whether the connection is HTTP or serial tunneling. This input operation times out in HTTP_Timeout_Msec if no lines are present. If the line does not identify as HTTP, it is declared as a serial tunneling connection and Ether_Connect_Status returns a value of PASSIVE_NON_WEB_CONNECTION which must be serviced by the application task. Note that if the connection is not identified as HTTP (because a leading GET substring is not found), the incoming bytes accepted by the Ether_Connection_Manager remain in the HTTP_Inbuf as an lstring (16-bit count followed by the data bytes) where they can be processed by the application program. In this case, the application program learns of the accepted passive connection by polling the Ether_Connect_Status routine; see its glossary entry. If the line identifies as HTTP, this HTTP_Server webserver routine is called by the Ether_Connection_Manager. It invokes HTTP_Parse_URL (see its glossary entry) to parse the URL (web address string) in place in the HTTP_Inbuf, isolating the "base URL" from the query field (if any), and un-escaping any escape sequences in the base URL to prepare it for string matching to the posted URLs. Note that each posted URL should start with a / character, and for completeness, a URL comprising a single / character should be in the table in case the user points a browser at the bare ip address of the wildcard. Then HTTP_Server compares the base URL to the entries in the autoserve array posted by HTTP_Add_Handler. If a match is found, the web service request is automatically fulfilled by dispatching the posted handler for the URL, and then the connection is closed by the Ether_Connection_Manager. If there is no match to routines posted by HTTP_Add_Handler, this function looks for a match to the URLs posted by HTTP_Add_GUI_Handler. If a match is found, the matching handler xcfa (with the modulenum packed into its most significant byte) is sent in the ether_gui_message mailbox. If the web based "remote front panel" feature is supported by the application, the application task must poll the Ether_Check_GUI routine to receive this

message when present. If there is no match found at all for the received base URL in the autoserve array, the routine pointed to by HTTP_Default_Handler_Ptr is executed to serve out the “404 Not Found” error webpage. Typically this is performed by HTTP_Default_Handler (see its glossary entry). In all of these cases, after the handler has executed, the Ether_Connection_Manager closes the HTTP connection that was opened by the remote web browser.

Notes on web handler functions posted by HTTP_Add_Handler: The handler function must accept one and only one 16-bit input parameter that specifies the EtherSmart/WiFi modulenum, and it must not return any parameters. For static web pages, the handler function typically sends the web page via a buffer in convenient chunks the size of the HTTP_Outbuf buffer or smaller until the file has been sent. Each handler can parse the query fields (if any) in the URL using the supplied parsing primitives: HTTP_To_Next_Field, HTTP_Fieldname_Ptr, HTTP_Fieldname_Count, HTTP_Value_Ptr, HTTP_Value_Count, HTTP_Unescape, and HTTP_Plus_To_Space. The user’s handler can call these routines to parse the query values, remove escape sequences and + characters from text entered in forms, and respond as needed by outputting static or dynamic text. Be careful: the handler is called from the Ethernet task, and any writes by an HTTP handler to a shared buffer such as ETHER_OUTBUF could conflict with simultaneous writes that are being performed by the application task. It is highly recommended that each service (HTTP, SMTP, serial tunneling) use its own buffers; use of HTTP_Outbuf is recommended for HTTP handler functions. When the HTTP handler returns, the Ether_Connection_Manager called by Ether_Service_Loop running in the Ethernet task closes the connection.

C: void **HTTP_Set_Inbuf** (xaddr xbufbase, int maxnumbytes, int modulenum)
4th: **HTTP_Set_Inbuf** (xbufbase \maxnumbytes\modulenum --)

Sets the 32-bit extended base address xbufbase and the size maxnumbytes of the HTTP_Inbuf default web service input buffer for the specified EtherSmart/WiFi module. Note that the allocated buffer size must be 2 bytes bigger than the maxnumbytes parameter; these 2 bytes provide room to store the 16- bit count at the start of the lbuffer. The Ether_Setup and WiFi_Setup functions initialize HTTP_Inbufsize to HTTP_INBUFSIZE_DEFAULT = 254, with an allocated buffer size of 256 bytes. See HTTP_Inbuf.

C: void **HTTP_Set_Outbuf** (xaddr xbufbase, int maxnumbytes, int modulenum)
4th: **HTTP_Set_Outbuf** (xbufbase \maxnumbytes\modulenum --)

Sets the 32-bit extended base address xbufbase and the size maxnumbytes of the HTTP_Outbuf default web service output buffer for the specified EtherSmart/WiFi module. Note that the allocated buffer size must be 2 bytes bigger than the maxnumbytes parameter; these 2 bytes provide room to store the 16- bit count at the start of the lbuffer. The Ether_Setup and WiFi_Setup functions initialize HTTP_Outbufsize to HTTP_OUTBUFSIZE_DEFAULT = 1022, with an allocated buffer size of 1024 bytes (1 Kbyte). See HTTP_Outbuf.

C: xaddr **HTTP_Status_Ptr** (int modulenum)
4th: **HTTP_Status_Ptr** (modulenum -- xaddr)

Returns the xaddress within the ether_info struct for the specified EtherSmart/WiFi module that holds a 32-bit quantity that is reserved for the user. The programmer can craft HTTP handler functions that write information into this location so that the foreground application program can monitor web server; recall that non-GUI web service responses are dispatched from the Ethernet task. The contents at HTTP_Status_Ptr are erased by Ether_Info_Init, WiFi_Info_Init and their callers as shown in the “Initialization” section of the categorized function list.

C: **HTTP_TEXT_HTML_CONTENT**
4th: **HTTP_TEXT_HTML_CONTENT**

A 16-bit constant that returns the value -1. This constant is passed as a content_identifier to HTTP_Put_Content_Type to indicate that the HTTP header should declare the content type as "text/html". This is the default content type for web pages.

C: **HTTP_TEXT_PLAIN_CONTENT**
4th: **HTTP_TEXT_PLAIN_CONTENT**

A 16-bit constant that returns the value 0. This constant is passed as a content_identifier to HTTP_Put_Content_Type to indicate that the HTTP header should declare the content type as "text/plain". Note that this content type does not support HTML format tags; typically, the HTTP_TEXT_HTML_CONTENT type is preferred for web pages.

C: xaddr **HTTP_Timeout_Msec_Ptr** (int modulenum)
4th: **HTTP_Timeout_Msec_Ptr** (modulenum -- xaddr)

Returns the xaddress within the ether_info struct for the specified EtherSmart/WiFi module that holds a 16-bit timeout for outgoing HTTP traffic in units of milliseconds. The maximum allowed timeout is 65,535 ms. This timeout is used by HTTP_Send_Buffer, HTTP_Send_LBuffer, HTTP_Send_2Buffers, HTTP_GUI_Send_Buffer, HTTP_GUI_Send_LBuffer, and HTTP_GUI_Send_2Buffers 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. See also HTTP_Get_Timeout_Msec_Ptr.

C: int **HTTP_To_Next_Field** (int modulenum)
4th: **HTTP_To_Next_Field** (modulenum -- numchars_advanced)

This routine advances the URL (Universal Resource Locator) field pointers to point to the next query field and returns the number of chars advanced. If 0 is returned, there are no more query fields in the URL. This function assumes that HTTP_Parse_URL has been automatically called by the webserver in the Ethernet task running Ether_Service_Loop. This routine is used during the handling of webservice requests to parse the query field of the URL. Each field is of the form:

fieldname=value

Within the current query field, use the functions HTTP_Fieldname_Ptr, HTTP_Fieldname_Count, HTTP_Value_Ptr, and HTTP_Value_Count to identify the fieldname and value parameters passed by the browser's GET command string. Let's assume that the handler for the URL string /form_entry.html is the one described in the demo code that accompanies this driver. Then when the "submit" button is pressed after the user fills in the form, the browser opens a connection and sends a request of the form:

GET /form_response.cgi?classification=man&name_id=tommy&color=blue

In this case the URL is defined as

/form_response.cgi

This is the string that is passed to HTTP_Add_Handler to associate the handler function with the URL. The text after the URL starting with the ? character is the query portion of the URL. Each "field" in the query comprises a fieldname followed by an = character followed by a field value. Fields are separated by the & character. When the handler function is first called, executing Http_Fieldname_Ptr returns the xaddress of the 'c' in 'classification' and the HTTP_Fieldname_Count returns 14, the count of 'classification'. HTTP_Value_Ptr returns the xaddress of the 'm' in 'man', and HTTP_Value_Count returns 3, the count of 'man'. A call to

HTTP_To_Next_Field advances the current field to after the next & character. Then Http_Fieldname_Ptr returns the xaddress of the 'n' in 'name_id' and the HTTP_Fieldname_Count returns 7, the count of 'name_id'. HTTP_Value_Ptr then returns the xaddress of the 't' in 'tommy', and HTTP_Value_Count returns 5, the count of 'tommy'. An additional call to HTTP_To_Next_Field advances the current field to after the next & character. Then Http_Fieldname_Ptr returns the xaddress of the 'c' in 'color' and the HTTP_Fieldname_Count returns 5, the count of 'color'. HTTP_Value_Ptr then returns the xaddress of the 'b' in 'blue', and HTTP_Value_Count returns 4, the count of 'blue'. Because the URL is now exhausted, an additional call to HTTP_To_Next_Field would result in HTTP_Fieldname_Count and HTTP_Value_Count returning 0, signalling that there are no additional query fields to be examined. See the glossary entry of HTTP_Imagemap function for information on how to handle an HTTP GUI "remote front panel" web request.

C: int **HTTP_Unescape** (xaddr xstring, int count)
4th: **HTTP_Unescape** (xaddr\count -- revised_cnt)

The input parameters specify the starting 32-bit xaddress and count of a substring in a URL (Universal Resource Locator string from a web browser). The unescape process replaces each HTTP 3-character escape sequence %hexhex (where hex is a hexadecimal digit) in the specified substring with its equivalent single ascii character corresponding to the 2-digit hexadecimal value, and shortens the URL field and its count by two chars for each escape sequence encountered. This routine is available for use by the programmer in parsing query subfields; it is automatically applied by HTTP_Parse_URL to the base portion of the URL. After performing the unescape and string compression, this routine writes blanks over any "orphaned" characters remaining after the URL (or after the query field, if present).

CAUTION: If you are unescaping a query field substring, and the result of the un-escape could introduce a reserved & or = character (the field and value delimiters), the URL parsing routines could get confused by the unescaped characters masquerading as valid delimiters. To solve this problem, make sure that you save the current HTTP_Fieldname_Ptr, HTTP_ Fieldname_Count, HTTP_Value_Ptr, and HTTP_Value_Count, and execute HTTP_To_Next_Field, before calling HTTP_Unescape for the saved query substring. This ensures that the parsing operations occur before the un-escape operation introduces any confusing delimiters that could corrupt the result. Note that the following characters are "reserved" in URLs and must be escaped (encoded as %hexhex):

; / ? : @ = &

The following characters are "unsafe" in URLs and should be escaped:

< > " # % { } | \ ^ ~ [ ] '

C: int **HTTP_URL_Base_Count** (int modulenum)
4th: **HTTP_URL_Base_Count** (modulenum -- count)

Returns the count of the URL from the first non-blank character after GET (pointed to by HTTP_URL_Ptr) until either the ? character that marks the start of the query field (if present), or the terminating blank (if no query field is present). The returned count includes the starting / character of the URL, and does not include the terminating ? or blank character. The base portion of the URL is the string that is matched to handlers posted by HTTP_Add_Handler and HTTP_Add_GUI_Handler. See the glossary entries for HTTP_Parse_URL and HTTP_Add_Handler for further information about URLs.

C: int **HTTP_URL_Full_Count** (int modulenum)
4th: **HTTP_URL_Full_Count** (modulenum -- count)

Returns the count of the full URL including query fields (if any), starting at the first non-space char (typically a / character) after the GET<space> keyword of the web request, and ending with the last non-blank character. The terminating blank of the URL is not included in the count. See the glossary entries for HTTP_Parse_URL and HTTP_Add_Handler for further information about URLs.

C: xaddr **HTTP_URL_Ptr** (int modulenum)
4th: **HTTP_URL_Ptr** (modulenum -- xaddr)
Returns the uncounted string xaddress of the first non-space char after the GET<space> keyword in the HTTP_Inbuf for the most recently received web connection. The character pointed to by xaddr is typically / (forward slash); hence, all URL's posted should start with the / character. See the glossary entries for HTTP_Parse_URL and HTTP_Add_Handler for further information about URLs.

C: int **HTTP_Value_Count** (int modulenum)
4th: **HTTP_Value_Count** (modulenum -- count)
Returns the count of the current query field after the = sign of the URL (Universal Resource Locator string) for the specified EtherSmart/WiFi module. The count does not include the leading = character, nor the following & or blank terminating delimiter. Returns 0 if the query field is not present or the URL is exhausted. This routine is used during the handling of webservice requests to parse the query field of the URL. When the handler associated with the incoming URL is first called, this function returns the fieldname count of the first query field after the ? (if present). To advance to the next field, call HTTP_To_Next_Field. Each field is of the form:

fieldname=value

Within the current query field, use the functions HTTP_Fieldname_Ptr, HTTP_Fieldname_Count, HTTP_Value_Ptr, and HTTP_Value_Count to identify the fieldname and value parameters passed by the browser's GET command string. Let's assume that the handler for the URL string /form_entry.html is the one described in the demo code that accompanies this driver. Then when the "submit" button is pressed after the user fills in the form, the browser opens a connection and sends a request of the form:

GET /form_response.cgi?classification=man&name_id=tommy&color=blue

In this case the URL is defined as

/form_response.cgi

This is the string that is passed to HTTP_Add_Handler to associate the handler function with the URL. The text after the URL starting with the ? character is the query portion of the URL. Each "field" in the query comprises a fieldname followed by an = character followed by a field value. Fields are separated by the & character. When the handler function is first called, executing Http_Fieldname_Ptr returns the xaddress of the 'c' in 'classification' and the HTTP_Fieldname_Count returns 14, the count of 'classification'. HTTP_Value_Ptr returns the xaddress of the 'm' in 'man', and HTTP_Value_Count returns 3, the count of 'man'. A call to HTTP_To_Next_Field advances the current field to after the next & character. Then Http_Fieldname_Ptr returns the xaddress of the 'n' in 'name_id' and the HTTP_Fieldname_Count returns 7, the count of 'name_id'. HTTP_Value_Ptr then returns the xaddress of the 't' in 'tommy', and HTTP_Value_Count returns 5, the count of 'tommy'. An additional call to HTTP_To_Next_Field advances the current field to after the next & character. Then Http_Fieldname_Ptr returns the xaddress of the 'c' in 'color' and the HTTP_Fieldname_Count returns 5, the count of 'color'. HTTP_Value_Ptr then returns the xaddress of the 'b' in 'blue', and HTTP_Value_Count returns 4, the count of 'blue'. Because the

URL is now exhausted, an additional call to HTTP_To_Next_Field would result in HTTP_Fieldname_Count and HTTP_Value_Count returning 0, signalling that there are no additional query fields to be examined. See the glossary entry of HTTP_Imagemap function for information on how to handle an HTTP GUI "remote front panel" web request.

C: xaddr **HTTP_Value_Ptr** (int modulenum)
4th: **HTTP_Value_Ptr** (modulenum -- xaddr)

Returns the 32-bit extended address of the first character after the = in the current query field of the URL (Universal Resource Locator string) for the specified EtherSmart/WiFi module This routine is used during the handling of webservice requests to parse the query field of the URL. The returned xaddress points to the character after the ? (first field only) or & (all subsequent fields). When the handler associated with the incoming URL is first called, this function points to the first query field after the ? (if present). To advance to the next field, call HTTP_To_Next_Field. Each field is of the form:

fieldname=value

Within the current query field, use the functions HTTP_Fieldname_Ptr, HTTP_Fieldname_Count, HTTP_Value_Ptr, and HTTP_Value_Count to identify the fieldname and value parameters passed by the browser's GET command string. Let's assume that the handler for the URL string /form_entry.html is the one described in the demo code that accompanies this driver. Then when the "submit" button is pressed after the user fills in the form, the browser opens a connection and sends a request of the form:

GET /form_response.cgi?classification=man&name_id=tommy&color=blue

In this case the URL is defined as

/form_response.cgi

This is the string that is passed to HTTP_Add_Handler to associate the handler function with the URL. The text after the URL starting with the ? character is the query portion of the URL. Each "field" in the query comprises a fieldname followed by an = character followed by a field value. Fields are separated by the & character. When the handler function is first called, executing Http_Fieldname_Ptr returns the xaddress of the 'c' in 'classification' and the HTTP_Fieldname_Count returns 14, the count of 'classification'. HTTP_Value_Ptr returns the xaddress of the 'm' in 'man', and HTTP_Value_Count returns 3, the count of 'man'. A call to HTTP_To_Next_Field advances the current field to after the next & character. Then Http_Fieldname_Ptr returns the xaddress of the 'n' in 'name_id' and the HTTP_Fieldname_Count returns 7, the count of 'name_id'. HTTP_Value_Ptr then returns the xaddress of the 't' in 'tommy', and HTTP_Value_Count returns 5, the count of 'tommy'. An additional call to HTTP_To_Next_Field advances the current field to after the next & character. Then Http_Fieldname_Ptr returns the xaddress of the 'c' in 'color' and the HTTP_Fieldname_Count returns 5, the count of 'color'. HTTP_Value_Ptr then returns the xaddress of the 'b' in 'blue', and HTTP_Value_Count returns 4, the count of 'blue'. Because the URL is now exhausted, an additional call to HTTP_To_Next_Field would result in HTTP_Fieldname_Count and HTTP_Value_Count returning 0, signalling that there are no additional query fields to be examined. See the glossary entry of HTTP_Imagemap function for information on how to handle an HTTP GUI "remote front panel" web request.

4th: **LCOUNT** (xLongstring_addr -- xaddr_of_first_char\ucount)

This function is only available in Forth, as C functions are limited to one return parameter. This function "unpacks" a counted longstring into its starting xaddress and count. The input parameter is a 32-bit extended address that points to a longstring comprising a 2-byte count

followed by the string contents. The output xaddress points to the first character in the string, and the output ucount is the 16-bit count of the string.
Implementation details: ucount is simply fetched from xLongstring_addr. xaddr_of_first_char is simply the xaddress that is 2 bytes greater than xLongstring_addr.
Kernel notes: This function is built into the V6.xx kernels, and is defined in this driver code for V4.xx kernels.

4th: **LPARSE** (xpointer_to_result_area\maxchars\eol\delimiter -- xLstring_addr)
This function is only available in Forth; C uses other methods to define strings. LPARSE moves lines of text from the input stream's terminal input buffer (TIB) to the memory area pointed to by the xpointer_to_result_area (typically DP or NP or VP). After the specified delimiter is found or after maxchars have been parsed (whichever comes first), a terminating null character is added and the xpointer_to_result_area is updated to point to the byte after the terminating null. As each line ending in the input stream is encountered, the specified eol (end of line) sequence is inserted into the result string. The eol parameter can contain 1 or 2 bytes; typical values are 0x0D (carriage return), 0x0A (linefeed), 0x0D0A (carriage return/linefeed), or 0x00 (null byte). If the eol parameter = -1, no eol bytes are stored in the string by this function. If the most significant (ms) byte of the eol parameter is non-zero, two bytes are stored in the buffer after the appended string: first the msbyte of eol, then the lsbyte of eol. LPARSE returns xLstring_addr, an 32-bit extended address that points to a 2-byte longstring count followed by the parsed text. The terminating null is not included in the stored count.
Kernel version notes: LPARSE is built into the V6.xx kernel, and is defined in this driver code for V4.xx kernels. There are two key differences between the versions. The V4.xx LPARSE starts parsing on the line following following the LPARSE keyword, ignoring the remainder of the line containing LPARSE. The V6.xx version of LPARSE, on the other hand, starts parsing immediately after the LPARSE keyword. In addition, the V6.xx version supports 1- or 2-byte delimiters, while the V4.xx version is limited to 1-byte delimiters.
Cautions: The xpointer_to_result_area parameter is typically DP or NP or VP; do not use HERE or NHERE or VHERE. The available memory for storing the result string should be at least 3 bytes larger than the specified maxchars to allow for the 2-byte count and the terminating null byte. The maximum value of maxchars = 65,532. The stored output string can cross page boundaries. If the specified delimiter character is a blank, then an end of line in the input stream is interpreted as a valid ending delimiter that will terminate the string. If this routine is terminated by maxchars being reached, the remainder of the input line on which the termination occurred will not be interpreted by the compiler; these orphaned characters are removed from the input stream. A maximum of 94 characters per line can be parsed by the V4.xx version of this routine.
Terminal note: To speed downloads, the QEDTerm program strips empty lines and lines starting with the \ comment character out of the download file by default. This can alter the content of strings appearing between LPARSE and the delimiter in your download file. To solve this problem, use the #nostripcomments directive alone on a line in your download file to turn off comment stripping, then define the string(s) using one or more LPARSE directives, and restore the comment stripping using the #stripcomments directive alone on a line. If your version of QEDTerm does not support these directives, download the latest version of QEDTerm from www.mosaic-industries.com.
Example of use: Let's say you want to create a multi-line string located in the dictionary area (which will be in non-volatile memory), with a carriage return and linefeed after every line in the stored string. There are no empty lines in the string, so we don't need to use the QEDTerm directives. We'll choose the } character as our ending delimiter, and we'll specify 0x400 (1 Kbyte) characters as the maximum string size. The longstring would be created as follows:

```
HEX
DP 400 0D0A ASCII } LPARSE
This is a test string
compiled by LPARSE!}
```

The result left on the stack is the xaddress of the longstring buffer containing a 2-byte count followed by the string contents.

C: **WIFI_CCMP_GROUP_ENCRYPT**
4th: **WIFI_CCMP_GROUP_ENCRYPT**

A 16-bit constant that returns the value 4. This constant can be passed to the WiFi_Security function as the group_encrypt_method parameter to specify the CCMP group encryption method that is associated with 802.11i/WPA2 security. See the glossary entry for WiFi_Security.

C: **WIFI_CCMP_PAIR_ENCRYPT**
4th: **WIFI_CCMP_PAIR_ENCRYPT**

A 16-bit constant that returns the value 4. This constant can be passed to the WiFi_Security function as the pairwise_encrypt_method parameter to specify the CCMP pairwise encryption method that is associated with 802.11i/WPA2 security. See the glossary entry for WiFi_Security.

C: int **WiFi_Check** (int modulenum)
4th: **WiFi_Check** (modulenum -- flag)

Returns a true flag if the specified modulenum has been marked at initialization time as a WiFi Wildcard by one or more of the functions WiFi_Module, WiFi_Info_Init, WiFi_Init, WiFi_Setup, WiFi_Setup_Default, or WiFi_Task_Setup. Returns false if the module has been initialized as an EtherSmart Wildcard. This low level function is typically not used by the end programmer.

C: int **WiFi_Encryption_Key** (xaddr key_buffer_xbase, int num_key_bytes, int modulenum)
4th: **WiFi_Encryption_Key** (key_buffer_xbase\num_key_bytes\modulenum--error)

For the specified Wifi module, installs into the WiPort record 9 flash the WiFi encryption key that has been stored at key_buffer_xbase with length num_key_bytes. 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 num_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 key_buffer_xbase. This function returns 0 if successful, or returns a nonzero flag if the operation was not successful. 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; see the glossary entries for WiFi_SSID and WiFi_Security. Devices that do not know the security key will in theory not be able to eavesdrop on the secure communications.

Key length description:

A passphrase is 8 to 63 printable ascii bytes; this routine ensures that a trailing zero (null delimiter) is stored after the passphrase. A passphrase can be used with WEP or WPA.

WEP64 hex key is 40 bits = ten hex digits, e.g. 123456789A

WEP128 hex key is 104 bits = 26 hex digits, e.g. 123456789ABCDEF0123456789A

WPA/TKIP key is 128 bits = 32 hex digits, e.g. 123456789ABCDEF0123456789ABCDEF0

Note:The key length parameter passed to the WiFi_Security function 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.

To use: After a restart, initialize the WiFi Wildcard using WiFi_Setup or one of its calling functions listed in the "Initialization" section of the categorized function list, and then invoke WiFi_Encryption_Key, passing it the base xaddress and size of the key which 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 record 8 via the functions Wifi_Security, WiFi_SSID, and WiFi_Options, and then execute Ether_XPort_Update; see their glossary entries for details. 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 (see WiFi_Task_Setup) and execute Ether_XPort_Defaults (see its glossary entry; it works for both EtherSmart and WiFi Wildcards). Remember to adjust the security settings on your PC and/or wireless access point, as all security parameters must match for communications to occur.

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

Implementation detail: This routine acts directly on the Lantronix hardware, not via the ether task, and so should be executed when there is no network activity being managed by the Ethernet task. This function suspends multitasking for several seconds to enter the monitor mode, resumes multitasking, moves the default block9 contents to Ether_Outbuf, writes the encryption key, writes block9 to the Lantronix device, then executes the monitor mode RS reset command to instantiate the new values into the Lantronix flash memory. The entire operation takes approximately 13 seconds, so please be patient.

C: void **WiFi_Info_Init** (xaddr xinfo_struct_base, xaddr xautoserve_array_base, int numrows,
xaddr xbuffer_area_base, xaddr xcommand_mailbox, xaddr xresponse_mailbox,
xaddr xgui_mailbox, uint ether_inbufsize, uint ether_outbufsize,
uint http_inbufsize, uint http_outbufsize, int modulenum)

4th: **WiFi_Info_Init** (xinfo_struct_base\ xautoserve_array_base\numrows
\xbuffer_area_base\xcommand_mailbox\xresponse_mailbox
\xgui_mailbox\ether_inbufsize\ether_outbufsize
\http_inbufsize\http_outbufsize\modulenum --error)

This function initializes a WiFi Wildcard; see Ether_Info_Init to initialize an EtherSmart Wildcard. Initializes the entries in the ether_info struct located at the specified xinfo_struct_base, and makes a table entry for the specified WiFi modulenum/ether_info pair so that the other driver functions can locate the xinfo_struct_base given the modulenum, and passes xautoserve_array_base and numrows to HTTP_Is_Autoserve_Array to allocate the autoserve array, then erases the autoserve array in RAM. Starting at the specified 32-bit xbuffer_area_base location which can be in paged or common RAM, allocates in order Ether_Inbuf having size specified by the ether_inbufsize parameter+2, Ether_Outbuf having size specified by the ether_outbufsize parameter+2, HTTP_Inbuf having size specified by the http_inbufsize parameter+2, and HTTP_Outbuf having size specified by the http_outbufsize parameter+2. The total number of bytes allocated for these buffers is:

8 + ether_inbufsize + ether_outbufsize + http_inbufsize + http_outbufsize

where the 8 + is for the 2-byte count at the beginning of each buffer. In other words, to allow for the 2-byte count to be stored before the content area of each buffer, the allocated size of each buffer is 2 bytes greater than the buffer contents size parameter that is passed to this routine. Stores a 16-bit zero into the first 2 bytes of each of these buffers to initialize each buffer count to zero. This function stores 0\0 into each of the 3 mailboxes ether_command, ether_response, and ether_gui_message specified by xcommand_mailbox, xresponse_mailbox, and

xgui_mailbox, respectively. Each mailbox is specified by a 32-bit extended address that points to a 32-bit location in common RAM (mailboxes cannot be located in paged RAM). This function returns a zero flag if there is no error. If WiFi_Info_Init detects that the specified size of the Ether_Outbuf or Ether_Inbuf is smaller than ETHER_MIN_BUFFER_SIZE = 320, it returns ERROR_BUFFER_TOO_SMALL = 0x80 and sets Ether_Error to this value (see Ether_Error). This is not a fatal error, but it is best to declare buffers that meet or exceed this minimum size so that all of the functions that rely on these buffers will work properly, including Ether_XPort_Defaults, Ether_XPort_Update, Ether_Ping_Request, Ether_Ping_Report, Ether_IP_Info_Request and Ether_IP_Info_Report. Note that the buffer sizes are available using the functions Ether_Inbuf, Ether_Outbuf, HTTP_Inbuf, and HTTP_Outbuf, and their sizes are available as Ether_Inbufsize, Ether_Outbufsize, HTTP_Inbufsize and HTTP_Outbufsize. This routine turns on the variables pointed to by HTTP_Enable_Ptr and Ether_Tunnel_Enable_Ptr. To disable one or both of these passive services, the program must turn them off explicitly by storing a zero after this routine (or its caller) executes. Initializes the contents of HTTP_Get_Timeout_Msec_Ptr to 5000 (a 5 second timeout for incoming HTTP requests), and initializes HTTP_Timeout_Msec_Ptr to 33000 (a 33 second timeout for outgoing web send operations). Installs the execution xaddress of HTTP_Default_Handler into the HTTP_Default_Handler_Ptr to serve out the “404 Not Found” error page. Sets Ether_Local_Port to its default value of 80 which allows auto-detection of incoming web and passive serial tunneling connections. Sets Ether_Internal_Webserver_Port to its default value of 8000; this is used to access the built-in web configuration tool. WiFi_Info_Init is typically called by WiFi_Init; see its glossary entry.
Implementation detail: WiFi_Info_Init calls Ether_Info_Init and then invokes WiFi_Module.

C: void **WiFi_Init** (xaddr xinfo_struct_base, xaddr xautoserve_array_base, int numrows,
xaddr xbuffer_area_base, xaddr xcommand_mailbox, xaddr xresponse_mailbox,
xaddr xgui_mailbox, uint ether_inbufsize, uint ether_outbufsize,
uint http_inbufsize, uint http_outbufsize, int modulenum)

4th: **WiFi_Init** (xinfo_struct_base\ xautoserve_array_base\numrows
\xbuffer_area_base\xcommand_mailbox\xresponse_mailbox
\xgui_mailbox\ether_inbufsize\ether_outbufsize
\http_inbufsize\http_outbufsize\modulenum -- error)

This function initializes a WiFi Wildcard; see Ether_Init to initialize an EtherSmart Wildcard. This is the fundamental initialization routine that must be invoked before accessing the UART or WiPort hardware on the WiFi Wildcard. This routine calls WiFi_Info_Init; see its glossary entry for a detailed description of its actions including initializing the ether_info struct, mailboxes, timeouts, and local port, and allocating the buffers. WiFi_Init initializes the UART chip, powers up the WiPort chip, starts the timeslicer and globally enables interrupts. This routine initializes the ether_service_module variable which specifies which module is controlled by the Ether_Service_Loop webserver and command processor running in the Ethernet task. (This routine does not initialize the ether_revector_module variable). This routine flushes the UART input buffers for 0.25 second if no connection is open. If there is an open WiFi connection, this routine disconnects it. WiFi_Init is typically called by WiFi_Setup; see its glossary entry.

C: void **WiFi_Module** (int modulenum)

4th: **WiFi_Module** (modulenum --)

Marks the specified module as a WiFi Wildcard by writing a pattern to a reserved RAM location. This low level function typically not used by the programmer, as it is automatically invoked by

the initialization functions WiFi_Info_Init, WiFi_Init, WiFi_Setup, WiFi_Setup_Default, and WiFi_Task_Setup. Once this function has been invoked, the specified modulenum will be treated as a WiFi Wildcard by the driver routines.

C: **WIFI_NO_SECURITY**
4th: **WIFI_ NO_SECURITY**

A 16-bit constant that returns the value 0. This constant can be passed to the WiFi_Security function to specify any or all of the following parameters: authenticate_flag, suite_method, pairwise_encrypt_method, group_encrypt_method, or pairwise_encrypt_method. Passing this constant disables the named security or authentication method. Note that "no security" is the default state when the WiFi Wildcard is shipped from Mosaic, or after executing Ether_XPort_Defaults. See the glossary entry for WiFi_Security.

C: void **WiFi_Options** (int tx_power_level, int tx_power_manager_flag, int tx_rate,
int tx_auto_rate_flag, int adhoc_flag, int adhoc_chan, int modulenum)
4th: **WiFi_Options** (tx_power_level\tx_power_manager_flag\tx_rate\tx_auto_rate_flag
\adhoc_flag\adhoc_chan\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 as explained below. The tx_power_level parameter is a 2-bit code in the range 0 to 3 that specifies the transmit power in dBm with the following meaning: 0 (the default) means 0 dBm; 1 means 6 dBm; 2 means 12 dBm, and 3 means 18 dBm. The tx_power_manager_flag enables transmitter power management if nonzero, and disables it if zero (the default). The tx_rate parameter is a 3-bit code in the range 0 to 7 that sets the transmit data rate in Mbps (Megabits per second) with the following meaning:

0 means 1 Mbps
1 means 2 Mbps
2 means 5.5Mbps
3 means 11 Mbps
4 means 18 Mbps (the default)
5 means 24 Mbps
6 means 36 Mbps
7 means 54 Mbps

The tx_auto_rate_flag enables automated control of transmission rate if nonzero (the default), and disables automated rate control if zero. A nonzero adhoc_flag configures the Wildcard for "ad hoc" mode, which means that two wireless devices communicate with each other. 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. The adhoc_chan is a 4-bit code with a valid range of 1 to 13. If the adhoc_flag is true, then the adhoc channel number is set to the specified value. In the United States and Canada, allowed ad hoc channels are between 1 and 11 inclusive.

To Use: After executing this function, assuming that you have initialized the Ethernet task (see 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. See also WiFi_SSID and WiFi_Security.

C: 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)

4th: **WiFi_Security** (authenticate_flag\suite_method\pairwise_encrypt_method \group_encrypt_method\key_length\passphrase_flag \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 as explained below. The default upon shipment from Mosaic or after executing Ether_XPort_Defaults is "no security", equivalent to specifying zeros for the authenticate_flag, suite_method, pairwise_encrypt_method, and group_encrypt_method input parameters. 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. 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. 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.

To Use: After executing this function, assuming that you have initialized the Ethernet task (see 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. See also WiFi_SSID and WiFi_Security. See the EtherSmart/WiFi User Guide for additional discussion or WiFi security parameters.

Alternative method to customize security: You can use setup mode via a telnet client or log into the config web site at IP:8000 (before setting up security) and use the web configuration interface to specify the desired security settings. Of course, once you enable security on any part of the WLAN, the same key and security type must be setup on all to enable communications.

C: uint **WiFi_Setup** (xaddr xbuffer_area_base, addr mailbox_base_addr, int modulenum)

4th: **WiFi_Setup** (xbuffer_area_base\ mailbox_base_addr \module -- numbytes)

This function initializes a WiFi Wildcard; see Ether_Setup to initialize an EtherSmart Wildcard. WiFi_Setup is a high level initialization routine that calls WiFi_Init with default buffer locations and sizes starting at the specified xbuffer_area_base, and returns the number of bytes allocated there. Also passes to WiFi_Init the addresses of three mailboxes in common RAM starting at mailbox_base (a 16-bit address). The mailboxes are ether_command at mailbox_base_addr, ether_response at mailbox_base_addr+4, and ether_gui_message at mailbox_base_addr+8. These mailboxes must be declared (allocated) before calling this function. Locates all required buffers, the ether_info struct, and the HTTP autoserve array in RAM starting at the specified xbuffer_area_base, and returns the number of bytes allocated. Note that xbuffer_area_base can be in paged RAM to conserve common RAM for other uses. Zeros the three required mailboxes ether_command, ether_response, and ether_gui_message in common RAM. Starting at the specified 32-bit xbuffer_area_base, allocates the ether_info struct, followed in order by the autoserve array, Ether_Inbuf (512 bytes including 2-byte count), Ether_Outbuf (512 bytes including 2-byte count), HTTP_Inbuf (256 bytes including 2-byte count), and HTTP_Outbuf (1024 bytes including 2-byte count). Initializes the UART chip, powers up the WiPort chip, starts the timeslicer and globally enables interrupts. This routine initializes the ether_service_module variable which specifies which module is controlled by the Ether_Service_Loop webserver & command processor running in the Ethernet task. (This routine does not initialize the ether_revector_module variable). This routine flushes the UART input buffers for 0.25 second if no connection is open. If there is an open WiFi connection, this routine disconnects it. WiFi_Setup is typically called by WiFi_Setup_Default; see its glossary entry.

C: uint **WiFi_Setup_Default** (int modulenum)
4th: **WiFi_Setup_Default** (module -- numbytes)

This function initializes a WiFi Wildcard; see Ether_Setup_Default to initialize an EtherSmart Wildcard. WiFi_Setup_Default is a high level initialization routine that calls WiFi_Setup with a default xbuffer_area_base. For controllers running V6.xx kernels, passes 0x178000 (address 0x8000 on page 0x17) as the xbuffer_area_base to WiFi_Setup. For controllers running V4.xx kernels, passes 0x034000 (address 0x4000 on page 3) as the xbuffer_area_base to WiFi_Setup. Returns the number of bytes allocated at xbuffer_area_base; it is less than 3 Kbytes. See WiFi_Setup for a detailed description of operation. WiFi_Setup_Default is typically called by WiFi_Task_Setup; see its glossary entry.

Note: WiFi_Setup_Default specifies the same memory map as Ether_Setup_Default; these functions cannot be invoked for more that one Wildcard on a single controller. If your application requires multiple EtherSmart or WiFi Wildcards (or some combination or the two), you must specify non-overlapping memory maps and use WiFi_Setup or a lower level initialization function to configure each Wildcard.

C: void **WiFi_SSID** (xaddr ssid_buffer_xaddr, int count, int modulenum)
4th: **WiFi_SSID** (ssid_buffer_xaddr \count\modulenum --)

For the specified WiFi Wildcard module, writes the specified WiFi SSID (“Service Set ID”) ssid_buffer_xaddr and the count (clamped to a maximum of 32 bytes) into the ether_info struct. The ssid_buffer_xaddr is the 32-bit base address of the first character of the string, and count is the number of bytes in the string. Assuming that you have initialized the Ethernet task (see WiFi_Task_Setup), you can instantiate the string into the WiPort flash after invoking this function by executing Ether_XPort_Update followed by Ether_Await_Response. The 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”. SSID’s are case sensitive.

C: void **WiFi_Task_Setup** (TASK* task_base_addr, int modulenum)
4th: **WiFi_Task_Setup** (task_base_addr\modulenum --)

This function initializes a WiFi Wildcard; see Ether_Task_Setup to initialize an EtherSmart Wildcard. WiFi_Task_Setup is a high level routine that performs a full initialization of the ether_info struct and mailboxes, and builds and activates an Ethernet control task to service the WiPort for the specified modulenum. Calls WiFi_Setup_Default which in turn calls WiFi_Setup with a default xbuffer_area_base. For controllers running V6.xx kernels, passes 0x178000 (address 0x8000 on page 0x17) as the xbuffer_area_base to WiFi_Setup. For controllers running V4.xx kernels, passes 0x034000 (address 0x4000 on page 3) as the xbuffer_area_base to WiFi_Setup. Less than 3 Kbytes is allocated in this buffer area. See WiFi_Setup for a detailed description of operation. WiFi_Task_Setup then builds a standard size task running the Ether_Service_Loop activation routine at the specified 16-bit task_base address in common RAM (task areas must be in common RAM). To use this routine, allocate a 1 Kbyte (1024 byte) task area in common RAM (see the demo program described in the User Guide for an example; GCC task areas are allocated by the linker and may be larger than 1K). Pass its 16-bit base address along with the WiFi Wildcard modulenum (that corresponds to the hardware jumper settings) to this routine to perform a complete initialization and start the Ethernet task to service the connections. See the demo code for a convenient WiFi_Task_Setup_Default routine that uses a pre-defined 1 Kbyte task area to call this function. Also see the demo code for a WiFi_Web_Demo function that hosts a demonstration website from the WiFi Wildcard.

Note: WiFi_Task_Setup specifies the same memory map as Ether_Task_Setup; these functions cannot be invoked for more that one Wildcard on a single controller. If your application requires multiple EtherSmart or WiFi Wildcards (or some combination or the two), you must specify non-overlapping memory maps and use WiFi_Setup or a lower level initialization function to configure each Wildcard.

C: **WIFI_TKIP_GROUP_ENCRYPT**
4th: **WIFI_TKIP_GROUP_ENCRYPT**

A 16-bit constant that returns the value 3. This constant can be passed to the WiFi_Security function as the group_encrypt_method parameter to specify the TKIP (Temporal Key Integrity Protocol) group encryption method that is used in conjuction with WPA (WiFi Protected Access) security. Typically used in association with the WPA security suite with encryption specified as “TKIP”. See the glossary entry for WiFi_Security.

C: **WIFI_TKIP_PAIR_ENCRYPT**
4th: **WIFI_TKIP_PAIR_ENCRYPT**

A 16-bit constant that returns the value 3. This constant can be passed to the WiFi_Security function as the pairwise_encrypt_method parameter to specify the TKIP (Temporal Key Integrity Protocol) pairwise encryption method that is used in conjuction with WPA (WiFi Protected Access) security. See the glossary entry for WiFi_Security.

C: **WIFI_WEP128_PAIR_ENCRYPT**
4th: **WIFI_WEP128_PAIR_ENCRYPT**

A 16-bit constant that returns the value 2. This constant can be passed to the WiFi_Security function as the pairwise_encrypt_method parameter to specify the WEP128 (Wired Equivalent Privacy-128 bit) pairwise encryption method. WEP128 uses a 104-bit key concatenated with a 24-bit initialization vector. See the glossary entry for WiFi_Security.

C: **WIFI_WEP64_PAIR_ENCRYPT**
4th: **WIFI_WEP64_PAIR_ENCRYPT**
A 16-bit constant that returns the value 1. This constant can be passed to the WiFi_Security function as the pairwise_encrypt_method parameter to specify the WEP64 (Wired Equivalent Privacy-64 bit) pairwise encryption method. WEP64 uses a 40-bit key concatenated with a 24-bit initialization vector. See the glossary entry for WiFi_Security.

C: **WIFI_WEP_GROUP _ENCRYPT**
4th: **WIFI_WEP_GROUP _ENCRYPT**
A 16-bit constant that returns the value 1. This constant can be passed to the WiFi_Security function as the group_encrypt_method parameter to specify the WEP (Wired Equivalent Privacy) group encryption method. Typically used in association with the WPA security suite with encryption specified as “TKIP plus WEP group keys”. See the glossary entry for WiFi_Security.

C: **WIFI_WEP_SUITE**
4th: **WIFI_WEP_SUITE**
A 16-bit constant that returns the value 1. This constant can be passed to the WiFi_Security function as the suite_method parameter to specify the WEP (Wired Equivalent Privacy) security suite method. WEP can be implemented as 64 bit encryption which uses a 40-bit key concatenated with a 24-bit initialization vector, or 128 bit encryption which uses a 104-bit key concatenated with a 24-bit initialization vector. A hex key or an 8 to 63 byte passphrase can be used to specify the key. See the glossary entry for WiFi_Security.

C: **WIFI_WPA2_SUITE**
4th: **WIFI_WPA2_SUITE**
A 16-bit constant that returns the value 3. This constant can be passed to the WiFi_Security function as the suite_method parameter to specify the WPA2 (WiFi Protected Access 2) security suite method that is associated with the 802.11I protocol. See the glossary entry for WiFi_Security.

C: **WIFI_WPA_SUITE**
4th: **WIFI_WPA_SUITE**
A 16-bit constant that returns the value 2. This constant can be passed to the WiFi_Security function as the suite_method parameter to specify the WPA (WiFi Protected Access) security suite method. WPA uses a 128-bit key that can be specified as a hex key or using an 8 to 63-byte passphrase. See the glossary entry for WiFi_Security.

4th: **XTERM/38400SSP** (--)
A do-nothing word used only when performing an interactive login using the “Putty” network terminal. The name of this function corresponds to the default printable characters sent by the Putty network terminal when the “Rlogin” mode is selected for communications with the EtherSmart/WiFi Wildcard. The presence of this name in the QED-Forth names list prevents an error message from being displayed when the Putty Rlogin connection is established. The Rlogin mode has the correct combination of lack of echo and treatment of end-of-line characters for clean operation with the QED-Forth monitor. To obtain the free Putty TCP/IP terminal program, type

```
putty
```

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

GUI Toolkit Functions for a Remote Front Panel

C: void **Globalize_TVars** (GUI_VARS * tvars_addr, page tvars_page)
4th: **Globalize_TVars** (tvars_address\tvars_page --)
Initializes a global variable to contain the 32-bit extended base xaddress of the tvars structure. This function must be called at initialization time for the GUI Toolkit of the Qscreen product in order to implement the web-based "remote front panel" feature. This function need not be called if other versions of the GUI Toolkit are used (QVGA Board, PDQScreen, etc.) This function is required for use with the Qscreen so that the functions Screen_To_Image, Simulate_Touch, and Simulated_Touch_To_Image will have the same stack picture reguardless of the GUI Toolkit being used.

C: int **Graphic_To_Image** (xaddr xgraphic, xaddr xbuffer, uint buffer_size, int format_id)
4th: **Graphic_To_Image** (xgraphic\xbuffer\buffer_size\format_id -- error)
Converts the graphic object at the 32-bit xaddress xgraphic into an image at the specified 32-bit RAM xaddress xbuffer. The number of bytes stored in xbuffer is limited to the specified buffer_size parameter. The first 4 bytes stored at xbuffer contain a 32-bit byte count, followed by the image data in the specified format. The default format_id is BMP_FORMAT, corresponding to the bitmap (bmp) format (you can use the constant HTTP_IMAGE_BITMAP_CONTENT). Future versions of the GUI Toolkit for color screens will support other formats which convert a color image to a monochrome bitmap image to save memory space and data transfer time. Images must be representable in 65,535 or fewer bytes. This limits full-size quarter-VGA color graphics to 4 bit-per-pixel color depth. This function returns one of the following error values: NO_ERROR, INVALID_IMAGE_FORMAT, BUFFER_SIZE_TOO_SMALL, or INVALID_GRAPHIC.
Note: After this routine executes, a standard "LBuffer" comprising a 2-byte count followed by the data is present at the xbuffer+2 xaddress.

C: int **Has_Screen_Changed** (void)
4th: **Has_Screen_Changed** (-- flag)
Reports the value of the boolean flag that tells whether the screen image has been updated. This flag is set by Screen_Has_Changed (see its glossary entry), and is cleared by Simulated_Touch_To_Image upon a successful image conversion. This flag is used by Simulated_Touch_To_Image to detect whether a time-consuming image conversion needs to be undertaken after the GUI handler function associated with a button press has executed. Most applications will not directly call this function, instead using the high-level function Simulated_Touch_To_Image to implement the remote front panel feature.

C: void **Screen_Has_Changed** (void)
4th: **Screen_Has_Changed** (--)

Sets a global variable that indicates something on the current screen has changed. If your application implements a web-based “remote front panel” capability, code this function into each user-defined GUI Toolkit button handler that modifies the displayed screen. The goal is to distinguish button handlers that simply darken and restore a button on the one hand, from button handlers that cause the contents of the screen to be different after the button press compared to before it. The flag set by this function is used by Simulated_Touch_To_Image to detect whether a time-consuming image conversion needs to be undertaken after the GUI handler function associated with a button press has executed. The flag set by this function is readable using Has_Screen_Changed, and is cleared by Simulated_Touch_To_Image upon a successful image conversion.

C: int **Screen_To_Image** (xaddr xbuffer, uint buffer_size, int format_id)
4th: **Screen_To_Image** (xbuffer\buffer_size\format_id -- error)

Converts the current contents of the displayed screen into an image at the specified 32-bit RAM xaddress xbuffer. The number of bytes stored in xbuffer is limited to the specified buffer_size parameter. The first 4 bytes stored at xbuffer contain a 32-bit byte count, followed by the image data in the specified format. The default format_id is BMP_FORMAT, corresponding to the bitmap (bmp) format (you can use the constant HTTP_IMAGE_BITMAP_CONTENT). Future versions of the GUI Toolkit for color screens will support other formats which convert a color image to a monochrome bitmap image to save memory space and data transfer time. Images must be representable in 65,535 or fewer bytes. This limits quarter-VGA images to either monochrome or 4 bit-per-pixel color depth. This function returns one of the following error values: NO_ERROR, INVALID_IMAGE_FORMAT, or BUFFER_SIZE_TOO_SMALL.

Most applications will not directly call this function, instead using the high-level function Simulated_Touch_To_Image to implement the remote front panel feature.

Note: After this routine executes, a standard “LBuffer” comprising a 2-byte count followed by the data is present at the xbuffer+2 xaddress.

QScreen Notes: Globalize_TVars must be called at initialization time before using this routine on the Qscreen product. The QScreen buffer_size parameter must be at least 4160 bytes (bmp file size) + 3840 rendered text buffer size = 8000 bytes minimum for the QScreen.

QVGA Board Notes: The buffer_size parameter must be at least 9668 bytes to support the QVGA screen.

PDQScreen Notes: The buffer_size parameter must be at least 9668 bytes to support the PDQScreen in monochrome mode, and must be at least 38524 bytes to support a 4-bits per pixel color screen.

C: int **Simulated_Touch_To_Image** (xaddr xbuffer, uint buffer_size, int format_id, int x, int y)
4th: **Simulated_Touch_To_Image** (xbuffer\buffer_size\format_id \x\y -- result)

To implement a web-based “remote front panel” for your instrument, call this function after a click on an “imagemap” clickable webpage screen image, and before sending the new or refreshed screen to a waiting browser. If the specified x and y coordinates (in units of pixels, relative to the upper left corner of the screen) are within the active area of a button on the current screen, this function draws the button's press graphic, draws the button's release graphic, and executes the button’s “press” handler function. If the button handler changes the screen, this routine invokes Screen_To_Image to create a bitmap version of the current screen located at the specified xbuffer extended address in RAM, and limited to the specified maximum buffer_size. If the user-defined button handler changes the screen image, it must invoke the Screen_Has_Changed function in the “press” button handler so that this function will be aware that the screen image must be regenerated. The default format_id is BMP_FORMAT,

corresponding to the bitmap (bmp) format (you can use the constant HTTP_IMAGE_BITMAP_CONTENT). Future versions of the GUI Toolkit for color screens will support other formats which convert a color image to a monochrome bitmap image to save memory space and data transfer time. Images must be representable in 65,535 or fewer bytes. This function returns one of the following error values: NO_ERROR, INVALID_IMAGE_FORMAT, BUFFER_SIZE_TOO_SMALL, or SCREEN_UNCHANGED, XY_NOT_IN_BUTTON.

Note that the SCREEN_UNCHANGED and XY_NOT_IN_BUTTON values can be tested for with a "less than 0" test; these results do not necessarily indicate an error condition, but may be useful in optimizing the application code. A typical web/GUI application program will test that the error result = NO_ERROR (0) and, if so, serve the updated image to the browser.

Note: After this routine executes, a standard "LBuffer" comprising a 2-byte count followed by the data is present at the xbuffer+2 xaddress.

QScreen Notes: Globalize_TVars must be called at initialization time before using this routine on the Qscreen product. The QScreen buffer_size parameter must be at least 4160 bytes (bmp file size) + 3840 rendered text buffer size = 8000 bytes minimum for the QScreen.

QVGA Board Notes: The buffer_size parameter must be at least 9664 bytes to support the QVGA screen.

PDQScreen Notes: The buffer_size parameter must be at least 9664 bytes to support the PDQScreen in monochrome mode, and must be at least 38524 bytes to support a 4-bits per pixel color screen.

C: xaddr **Simulate_Touch** (int x, int y)

4th: **Simulate_Touch** (x\y -- handler_xcfa)

If the specified x and y coordinates (in units of pixels, relative to the upper left corner of the screen) are within the active area of a button on the current screen, draws the button's press graphic, draws the button's release graphic, and returns the button's press handler (if present). Otherwise returns a 32-bit zero. This function can be used in conjunction with an "imagemap" clickable web image of the screen to implement a web-based "remote front panel" for the instrument. Most applications will not directly call this function, instead using the high-level function Simulated_Touch_To_Image to implement the remote front panel feature.

QScreen Notes: Globalize_TVars must be called at initialization time before using this routine on the Qscreen product; Globalize_TVars is not needed for other platforms.