Winford Engineering ETH32 API Version 2 Documentation

Table of Contents

Winford Engineering ETH32 API Version 2 Documentation

Overview

The ETH32 API (Application Programming Interface) is a set of programming libraries provided by Winford Engineering to make the control of the ETH32 I/O device easier for customers. You can use the API in the applications you develop so you don't have to get involved in all the details of communicating with the ETH32. This greatly increases the speed of development and reduces the opportunities for bugs and errors. Although Winford Engineering recommends using the API in your application development, it is not required. The communication protocol used by the ETH32 is fully documented (in another document), allowing you to directly communicate with it over your own TCP/IP connection.

Platform and Language Support

The ETH32 API is available for the following platforms:

- Microsoft Windows 95, NT4, 98, 98SE, Me, 2000, XP, Vista, Windows 7

  A 64-bit version of the ETH32 API DLL is included for 64-bit Windows platforms, in addition to the standard 32-bit version.

- Linux

The following programming languages are supported:

- C
- C++
- Microsoft Visual Basic 6 and .NET
- Microsoft Visual Studio .NET Languages

Thread Safety

The ETH32 API is thread-safe, meaning that if your application is multithreaded, it is permissible to have multiple threads simultaneously calling API functions on the same handle (or object). The API itself is also multithreaded, creating and managing its own threads for each device connection. For the most part, this is transparent to you, the programmer. The only area you need to be aware of it is in event handling. If you use a callback function to receive events, your callback function will execute in a thread created by the API. Alternately, if you want to receive events without involving another thread, you can use one of the other event handler mechanisms.

In Visual Basic 6, multithreading can be very troublesome. For this reason, the VB 6 Eth32 class implements event handling internally without using extra threads or callback functions. From the programmer's perspective, you simply need to write an event routine and the rest is automatically taken

care of within the class.

In Visual Studio .NET languages, multithreading is fully supported, so the Eth32 .NET class implements event handling with a separate thread. This is all handled automatically within the class, so you simply need to write your own event routine. The advantage of being implemented with a separate thread is that events can be received and processed at the same time that the rest of your application is tied up with other tasks. Because it is executed in a separate thread, you must be careful that any operations you perform within your event routine are thread safe.

Installation

Since the ETH32 is a network device and all communication is done through TCP/IP sockets, no system-level device driver needs to be installed. The only files that are necessary are as follows:

- When writing and compiling your code: Header files, library files, or class source code, depending on the programming language.
- When running or distributing your compiled program: The eth32api.dll file (or the shared library on Linux) is needed. On Windows systems, eth32api.dll needs to either be in the same directory as your executable or in the system directory, for example on Windows XP, C:\Windows\System32. Programs created with a Microsoft .NET language also require the Eth32.dll assembly file.

Automatic Installation

An installation program is provided for Windows and Linux platforms that automatically installs the required system files. Programming language support files (headers, class source code, etc.) are not installed into any system directories since these should usually be copied into the project directory of any project that requires them. See the following sections for more information, depending on your programming language:

- C/C++ Getting Started
- .NET Languages Getting Started
- Visual Basic 6 Getting Started

Windows

Windows users are encouraged to use the automatic installation program to install all of the available components on their system. This includes the ETH32 API, sample applications, documentation, and the ETH32 network configuration utility. To do this, simply run the eth32_install.exe file located on the ETH32 product CD.

For Network Administrators needing to perform an unattended (silent) installation, the eth32_install.exe program may be run with the /S switch (case sensitive). The /D switch may also optionally be passed to override the default installation directory. If used, the /D option must be the last option and not contain any quotes, even if the path contains spaces, as in the example below:

```
eth32_install.exe /S /D=C:\Program Files\winford\eth32
```

Linux

Linux users are encouraged to run the install.sh script to install the ETH32 API onto their system. To do this, follow these simple steps:

- Log in as root.
- Depending on your system configuration, you may need to mount the CD
- Run the following commands:

```
cd /(path to cd drive)/api/linux
./install.sh
```

This script does not copy any example programs or documentation onto your system (they are available directly from the CD). For your information, it performs these tasks:

- Copies the ETH32 API libraries (libeth32.*) into /usr/lib/
- Configures the symbolic links to the shared library which are required for proper compile-time and run-time linking.
- Copies the eth32.h header file into /usr/include/

Topics

Although the ETH32 API is supported on several programming languages and separate documentation is included for some programming languages later in this document, most of the basic topics that should be understood are common among the languages. In other words, this section will help you to understand which functions need to be called for certain tasks, and the later sections will provide further details about calling the functions from your programming language.

Connections

The first thing your application must do in order to use the ETH32 device is create a connection to the device. A connection must be made before any other functions or members of the API are called, unless otherwise specified.

After your application has finished using the ETH32 device, it should be sure to disconnect the connection to the ETH32 device. The ETH32 device has a relatively small number of simultaneous connections available, so leaving a connection open unnecessarily could in some cases prevent other applications from opening a connection. When your application exits, the operating system will automatically close any open connections. However, Visual Basic 6 programmers should see the Remarks section of the Disconnect Method for precautions.

You may connect and disconnect from the device as often as you'd like. The decision of when to do this should be made depending on the type of application you are creating. If your application only performs an operation on the ETH32 once in a while and has long periods of inactivity, you may decide to connect and disconnect each time. However, most applications will simply connect once and remain connected until the application exits or it is completely finished using the ETH32. That is the simplest approach in most cases.

If at any point in your code, you have a connection but would like to verify that data is still being transferred over the connection properly, you may do so using the "*verify connection*" function. This sends a small command over your existing connection, which the ETH32 device will simply return without performing any other operation. If for any reason the data isn't returned within the timeout period, the function will return or generate an error.

In .NET languages as well as Visual Basic 6, the class provides a *Connected* property to indicate whether the object currently holds a connection to an ETH32 device. This property does not actually verify the integrity of the connection. When true, it simply indicates that a connection has been successfully created and that *Disconnect* has not been called since then. This is often useful during application shutdown in determining whether the *Disconnect* method needs to be called in order to free resources.

Note that the ETH32 libraries allow your application to simultaneously use as many ETH32 devices as resources allow. In certain circumstances, you may even want to open two connections to the same ETH32 device from within one application.

Almost all of the resources of the ETH32 are shared among its connections and are not affected by connections being opened or closed. So, for example, opening a new connection to the ETH32 does not affect the direction registers of any ports. One notable exception is that each connection individually

selects which event notifications it will receive. Therefore, each new connection starts out with all event notifications disabled.

Applicable Functions

Task	C / C++	.NET Languages	Visual Basic 6
Open connection	eth32_open	Connect Method	Connect Method
Verify connection	eth32_verify_connection	VerifyConnection Method	VerifyConnection Method
Determine if object is connected		Connected Property	Connected Property
Disconnect	eth32_close	Disconnect Method	Disconnect Method

Digital I/O

The I/O pins of the ETH32 are grouped into what are called *ports*. There are four ports 8-bit ports (eight I/O lines per port) and two 1-bit ports. Please see the ETH32 user manual for further description of the ports and the pinout of the connectors.

Each digital I/O pin can be individually configured as an input or output pin. Each port has a *direction register* which controls which of its bits are inputs and which are outputs. Functions are provided that can read or write the entire 8-bit direction register of a port or alternatively one bit at a time. When a direction register for a port is modified, that setting remains in effect until either you change it again, you reset the device, or the device is powered off. At powerup or reset, all I/O pins are configured as inputs.

The input value of a port indicates the status (high or low) of each bit of the port. If a bit is in output mode, the input value is still available and will be the same as the output value (with the possible exception of the pin being shorted or loaded much more than it should be).

The output value of a port controls the output voltage of the port's pins when the pins are in output mode. The output value also has a second purpose. When the pins are in input mode, any 1-bit in the output value will enable a weak pullup resistor on that pin. A 0-bit will disable the pullup resistor. A function is also provided that can read back the output value from the ETH32 device, regardless of whether the pins are in input mode or output mode.

Applicable Functions

Task	C / C++	.NET Languages	Visual Basic 6
Write direction register	eth32_set_direction eth32_set_direction_bit	SetDirection Method SetDirectionBit Method	SetDirection Method SetDirectionBit Method
Read direction register	eth32_get_direction eth32_get_direction_bit	GetDirection Method GetDirectionBit Method	GetDirection Method GetDirectionBit Method
Write output value	eth32_output_byte eth32_output_bit	OutputByte Method OutputBit Method	OutputByte Method OutputBit Method
Read back output value	eth32_readback	Readback Method	Readback Method
Read input value	eth32_input_byte eth32_input_bit	InputByte Method InputBit Method	InputByte Method InputBit Method

Special-Purpose Digital I/O

There are a few special digital I/O functions that are implemented directly on the ETH32 that may be useful in certain situations. Firstly, a *pulse bit* command allows you to rapidly pulse one of the output pins a specified number of times. This can be useful in initialization of an external device, such as a counter chip, if a line needs to be clocked a specific number of times.

Secondly, the ETH32 implements a command that allows a port's input value to be read repeatedly until two successive reads match. This can be useful if you are reading a multi-bit value such as a counter chip. Without this functionality, there would be a slight chance of reading the value during a transition period and obtaining an invalid value. By requiring that two successive reads match, that possibility is eliminated.

Applicable Functions

Task	C / C++	.NET Languages	Visual Basic 6
Pulse bit	eth32_pulse_bit	PulseBit Method	PulseBit Method
Successive read	eth32_input_successive	InputSuccessive Method	InputSuccessive Method

Analog Inputs

There are a few more steps involved in dealing with analog inputs than digital inputs. First, the entire Analog to Digital Converter (ADC) portion of the ETH32 device can be enabled or disabled by you at any time. In order for analog values to be read, you must enable the ADC.

You must also ensure that the ETH32 device is configured to use an appropriate analog voltage reference. Whenever an analog conversion is performed by the ADC, this voltage is used as a reference voltage representing the highest possible value. In other words, the readings are scaled so that a 0V signal on an input channel will give the lowest reading (0) and an input voltage equal to the reference voltage will give the highest reading (1023). Please see the *eth32_input_analog* function or *InputAnalog* method description for a mathematical representation of how the analog readings are obtained. The analog voltage reference

may be supplied by you on one of the pins of the ETH32 (referred to as the external voltage reference) or it may be obtained from one of the internal voltages. The powerup default is the external voltage reference.

The readings from the eight analog pins may be interpreted in different ways, all of which are completely selectable by you. First, you should understand the terms *single-ended* and *differential*. A single-ended input yields a reading representing the input voltage with respect to ground. A differential input consists of two input lines and the reading represents the voltage difference between those two lines. The value of the reading is not affected by the voltage of either signal with respect to ground, only the difference between the two voltages. However, please note that the signals are not electrically isolated from ground and must remain within the specified voltage limits of the input signals.

In this documentation, each of the available possibilities for interpreting the analog signals is referred to as a *physical channel*. For example, a single-ended input from bit 0 is one physical channel, a single-ended input from bit 1 is another physical channel, and the differential between bit 0 and bit 1 is another physical channel. In all, there are 32 physical channels, which are listed later in this document with the function for configuring analog channel assignments.

The ETH32 uses the concept of a *logical channel* to allow you to specify which of the physical channels should be continuously updated on the device and potentially monitored for event thresholds. There are eight logical channels. Each logical channel can be assigned by you to obtain its reading from any arbitrary physical channel. At powerup, the eight logical channels default to obtaining their readings from the eight single-ended physical channels. If your application requires using a differential channel, you will need to reassign one of the logical channels to obtain its reading from the desired differential channel.

When these settings have been configured appropriately for your application, you are ready to begin reading in analog values.

Applicable Functions

Task	C / C++	.NET Languages	Visual Basic 6
Enable/disable ADC	eth32_set_analog_state eth32_get_analog_state	AnalogState Property	AnalogState Property
Configure voltage reference	eth32_set_analog_reference eth32_get_analog_reference	AnalogReference Property	AnalogReference Property
Configure channel assignments	eth32_set_analog_assignment eth32_get_analog_assignment	AnalogAssignment Property	AnalogAssignment Property
Obtain analog readings	eth32_input_analog	InputAnalog Method	InputAnalog Method

Events

One of the powerful and useful features of the ETH32 is its event monitoring capabilities. In a nutshell, event monitoring allows the ETH32 to monitor different input signals and send a notification to your application when that signal has changed or met your criteria. Since the monitoring is constantly performed directly by the ETH32, it provides a much better alternative to polling. It provides faster response, is very efficient with network traffic and CPU resources, and is typically much easier to implement in your application.

Notifications of specific events are enabled and disabled on a per-connection basis on the ETH32. This means that each connection may individually select which events it wants to be notified of. This also provides efficient use of network traffic and the processing time of the ETH32. When events are sent from the ETH32 device, they are received on the PC side by the ETH32 API and then passed to your application, allowing your application to react accordingly.

Handling Events in Your Application

Handling events in your application is very easy although there are a few steps that must be performed first depending on your programming language. Essentially you must set up the mechanism that allows the ETH32 API to pass event information to your application whenever events occur.

.NET Languages and Visual Basic 6

In .NET languages and Visual Basic 6, events are handled by your application in a manner very similar to the way that Click events of buttons are handled. In other words, you must write an event handler routine that will be automatically called whenever events occur. If events occur faster than your code is processing them, they will be held in an internal queue. Class members are provided that allow you to configure the maximum size of the queue, to configure the queue behavior if the queue ever becomes full, and to retrieve the current number of events waiting in the queue. Events in the queue cannot be retrieved by calling a member function, but rather they are automatically passed to your event handler routine.

C/C++

In C/C++, there are two separate ways of receiving event data and/or being notified of new events. First, there is an *event handler* mechanism that you may configure to one of the predefined mechanisms, either a callback or a Windows message. The callback mechanism causes the API to call a function that you have written and pass in all of the information about the event each time an event fires. The Windows message mechanism sends a configurable Windows message to a window that you specify. The Windows message notifies you that an event has occurred, but does not include the information about the event. The second way of receiving event data is the *event queue*. If enabled, each event is added to the queue along with its event information. Events are stored in the queue until you have a chance to retrieve them. The event queue can be used independently of the event handler, but using them together can be very appropriate for the Windows message event handler. In other words, a Windows message can notify you that an event has occurred, which prompts you to retrieve the event information from the event queue.

The C/C++ function eth32_dequeue_event allows you to efficiently wait for new events to arrive if the event queue is empty. This capability can be used to synchronously wait for and process events, which can be desirable in some situations. Note that since the *event handler* and the *event queue* are independent of each other, both will receive a copy of each event that occurs if they are both enabled at the same time. For example, if the event handler is configured with a callback function and the event queue is enabled, both will receive a copy of the event information for each event that fires.

Applicable Functions and Information

Task	C / C++	.NET Languages	Visual Basic 6
Create event handler	Event Callback Function eth32_set_event_handler eth32_get_event_handler	Event Handler	Event Handler
Configure event queue	eth32_set_event_handler eth32_set_event_queue_config eth32_get_event_queue_status	EventQueueLimit Property EventQueueCurrentSize Property EventQueueMode Property	EventQueueLimit Property EventQueueCurrentSize Property EventQueueMode Property
Empty event queue	eth32_empty_event_queue	EmptyEventQueue Method	EmptyEventQueue Method
Retrieve event from queue	eth32_dequeue_event	Not applicable	Not applicable

Digital Events

Digital events are used to monitor the input value of a port, a bit within a port, or both. When enabled, a digital event sends a notification whenever the monitored bit or bits change value. Note that although digital events are typically used with bits that are in input mode, this is not enforced. If a digital event is enabled on a bit in output mode, it will fire an event every time you change the output value.

Digital events are supported on all bits of the ETH32's four 8-bit ports. Events may be enabled on any or all of these bits simultaneously. In order to achieve the fastest event monitoring speeds and use network traffic efficiently, you should only enable events on the bits where they are needed.

There are two types of digital events: *port events* and *bit events*. Port events monitor the entire 8-bit value of a port and notify you whenever that 8-bit value changes. Bit events monitor a single bit and notify whenever that bit changes.

Applicable Functions

Task	C / C++	.NET Languages	Visual Basic 6
Enable event notification	eth32_enable_event	EnableEvent Method	EnableEvent Method
Disable event notification	eth32_disable_event	DisableEvent Method	DisableEvent Method

Analog Events

Analog event monitoring is a little more involved than digital events. Analog events are based on event thresholds (also called an *event definition*), which are completely configurable by you. An event definition consists of a *lo-mark* and a *hi-mark*. These are two values which are configured within the ETH32 and allow the ETH32 to assign one of two values, low or high, to the event. When the analog reading is <= the lo-mark, the event will be considered low. When the analog reading is >= the hi-mark, the event will be considered high. When the reading is in between the two marks, the event keeps its previous state, allowing hysteresis to be built into the event. The ETH32 sends an event notification whenever the event state changes from low to high or from high to low.

The ETH32 allows a total of 16 analog event definitions to be in effect. These are organized into two *banks*, each having one analog event definition per *logical channel* (logical channels are described in the Analog Inputs section). This means that there are two analog event definitions per logical channel. Be aware that analog event definitions remain in effect on a logical channel even if you reassign the logical channel to obtain its reading from a different physical channel.

The lo-mark and hi-mark are specified as 8-bit numbers, while the analog readings are 10-bit numbers. The lo-mark and hi-mark specify the eight Most Significant Bits (MSB) of the analog reading. If you know the threshold you would like to set, represented as a 10-bit number, you can obtain the eight MSB's by doing an integer division by 4. There are no restrictions on the permissible values of the lo-mark and hi-mark other than that they must be 8-bit numbers (range 0-255) and that the lo-mark must be at least one less than the hi-mark.

The analog event definition banks are shared among all connections. However, the event notifications generated from the event definitions are enabled and disabled on a per-connection basis (just like digital events). Note that setting an analog event definition does not automatically enable receiving its event notifications - that must be done separately.

Applicable Functions

Task	C / C++	.NET Languages	Visual Basic 6
Set event definition	eth32_set_analog_eventdef	SetAnalogEventDef Method	SetAnalogEventDef Method
Get event definition	eth32_get_analog_eventdef	GetAnalogEventDef Method	GetAnalogEventDef Method
Enable event notification	eth32_enable_event	EnableEvent Method	EnableEvent Method
Disable event notification	eth32_disable_event	DisableEvent Method	DisableEvent Method

Counter Events

The ETH32 also supports a few events on its counters (for information on counters, see the Counters section below. There are two types of counter events. The first type is a *rollover event*, which sends an event notification whenever the counters value rolls over to zero. This type of event is supported on both counters. Note that the point at which the counter rolls over is configurable, but regardless, the event notification will be sent when the rollover occurs.

The second type of counter event is a *threshold event*. This type is only supported on counter 0. With this event, an event notification is sent whenever the counter's value exceeds a threshold that you have defined. There are no side-effects on the counter value from this type of event.

Applicable Functions

Task	C / C++	.NET Languages	Visual Basic 6
Configure counter rollover	eth32_set_counter_rollover eth32_get_counter_rollover	CounterRollover Property	CounterRollover Property
Configure counter threshold	eth32_set_counter_threshold eth32_get_counter_threshold	CounterThreshold Property	CounterThreshold Property
Enable event notification	eth32_enable_event	EnableEvent Method	EnableEvent Method
Disable event notification	eth32_disable_event	DisableEvent Method	DisableEvent Method

Counters

The ETH32 device includes two digital counters, which are useful for counting the number of pulses that occur on an I/O line. Counters are implemented directly in hardware and are therefore able to catch faster pulses than events are capable of catching (since they are implemented in ETH32 firmware).

Each counter can be independently enabled or disabled and configured to increment its count on either the falling edge of the input signal (transition from high to low) or the rising edge (transition from low to high). When a counter is disabled it simply means that its count will not increment regardless of the input signal. The counter's value can be read at any time. The counter's value may also be written to a specific value, which may be necessary during initialization.

Counter 0 is a 16-bit counter, while counter 1 is an 8-bit counter. Both counters allow you to configure a rollover value. When a counter reaches the rollover value, the next increment of the counter will reset the counter's value back to zero. The rollover values default to the maximum values of the counters on powerup or reset.

Applicable Functions

Task	C / C++	.NET Languages	Visual Basic 6
Enable/disable counter, configure edge	eth32_set_counter_state eth32_get_counter_state	CounterState Property	CounterState Property
Read/write counter value	eth32_get_counter_value eth32_set_counter_value	CounterValue Property	CounterValue Property
Configure counter rollover	eth32_set_counter_rollover eth32_get_counter_rollover	CounterRollover Property	CounterRollover Property

Pulse Width Modulation Outputs

The ETH32 includes two Pulse Width Modulation (PWM) output channels. PWM channels allow you to continuously output a configurable square wave pattern.

One typical use of a PWM signal is to provide variable speed control in a motor circuit. Like all outputs of the ETH32, the PWM outputs are low-current logic level outputs. Therefore in order to drive a load like a motor or other device, you will need to implement an appropriate transistor or FET circuit. With a PWM output, the transistor/FET will always be either fully off or fully on. Instead of controlling motor speed by altering voltage, the PWM signal alters the *duty cycle* of the motor. For example, the PWM signal may be configured to output a 10KHZ wave. If the PWM output is on for 75% of each wave cycle, the motor will run at a slower speed than if the PWM output is on for 100% of each cycle.

Some other typical uses of a PWM signal include more efficiently driving LED's and mechanical relays. For example, in the case of a mechanical relay, the signal can be briefly set to 100% duty cycle in order to pull the relay closed and then reduced to hold the relay closed, thereby reducing the current used in holding the relay closed.

Functions are provided by the API that make configuring the PWM channels very straightforward (eth32_set_pwm_parameters or SetPwmParameters). In addition, functions are provided that allow you to individually control several aspects of the PWM outputs. For those, you'll need a brief overview of the internal workings of the PWM outputs.

The PWM outputs are internally implemented using a 16-bit digital counter. Both outputs share the same counter. The counter is clocked at a rate of 2MHZ and the clock may be enabled or disabled at any time. For each output cycle, the square wave is created as follows: When the counter starts out at zero, the output signal is initially high. The counter clocks up and hits a configurable threshold at which point the output signal is set low. The counter continues clocking until it reaches the configurable rollover point, which resets the counter back to zero, sets the output signal high, and starts the entire cycle over again.

The rollover point of the counter is called the *base period* since it determines the length of each square wave output cycle. The threshold point is called the *duty period* since it determines how long the output will be on for each cycle.

Applicable Functions

Task	C / C++	.NET Languages	Visual Basic 6
Set PWM parameters (user friendly)	eth32_set_pwm_parameters	SetPwmParameters Method	SetPwmParameters Method
Get PWM parameters (user friendly)	eth32_get_pwm_parameters	GetPwmParameters Method	GetPwmParameters Method
Enable/disable PWM clock	eth32_set_pwm_clock_state eth32_get_pwm_clock_state	PwmClockState Property	PwmClockState Property
Enable/disable PWM channel	eth32_set_pwm_channel eth32_get_pwm_channel	PwmChannel Property	PwmChannel Property
Configure base period	eth32_set_pwm_base_period eth32_get_pwm_base_period	PwmBasePeriod Property	PwmBasePeriod Property
Configure duty period	eth32_set_pwm_duty_period eth32_get_pwm_duty_period	PwmDutyPeriod Property	PwmDutyPeriod Property

Built-in LEDs

The ETH32 includes two user-controllable LEDs on its front panel. The LEDs are designated as LED 0 and LED 1. They can easily be turned on or off independently. You may also read back the status of an LED to determine whether it is currently on or off.

Applicable Functions

Task	C / C++	.NET Languages	Visual Basic 6
Set/get LED status	eth32_set_led eth32_get_led	Led Property	Led Property

Product Identification

Each ETH32 device has a unique serial number assigned to it. Besides being printed on the device itself, the serial number can be retrieved from the device using functions provided by the API.

Functions are also provided to retrieve the version number of the firmware running on the device.

Lastly, each device has a product type ID which can be retrieved. This number is identical for almost all ETH32 devices and there is typically little need to retrieve or inspect it. However, if an ETH32 device is supplied with customized/special features, this number will reflect that.

Applicable Functions

Task	C / C++	.NET Languages	Visual Basic 6
Get serial number	eth32_get_serialnum eth32_get_serialnum_string	SerialNum Property	SerialNum Property
Get firmware version	eth32_get_firmware_release	FirmwareMajor Property FirmwareMinor Property	FirmwareMajor Property FirmwareMinor Property
Get product type ID	eth32_get_product_id	ProductID Property	ProductID Property

Timeouts and Errors

One potential source of errors when using the ETH32 device is from network problems. When the ETH32 is used on a Local Area Network (LAN), these kinds of problems should typically be quite small, but if the ETH32 is used over the internet, the probability of dropped packets or broken connections will increase. The ETH32 uses the TCP/IP protocol for all communications. TCP/IP is a robust, time-tested standard that includes data buffering and the provision for retrying data transmission, allowing connections to recover from lost or corrupted packets.

One problem that may occur due to network problems is a delay in receiving a response back from the ETH32. To deal with that the API includes a timeout setting. The timeout setting applies to any function that queries the ETH32 for data and waits for the response. For example, it would apply to reading the state of an LED, but it would not apply to setting the state of an LED. The timeout setting specifies a maximum amount of time the API should wait for a response. If the timeout period is ever exceeded, an error is generated or returned by the API. When a timeout occurs, it does not necessarily mean that the connection has been completely broken. The TCP/IP retransmissions may still recover from the error.

If communication is completely cut off, further requests will continue to time out until the TCP/IP stack of your operating system decides to give up and close the connection. After that happens, any further API calls that communicate with the ETH32 in any way will immediately fail with a network error.

Besides timeouts and broken connections, other errors include things such as specifying a value for a setting that is outside of the allowable limits, internal API errors, and others.

The way your application is notified of errors depends on the programming language you are using. Please see the Error Handling section for your programming language below. When errors are provided, an error code is included to identify the cause of the error. The error codes and their meanings are included in this document. The API also provides a function which will translate an error code into a string that briefly describes the error.

Applicable Functions and Information

Task	C / C++	.NET Languages	Visual Basic 6
Configure API Timeout	eth32_set_timeout eth32_get_timeout	Timeout Property	Timeout Property
Translate error codes	eth32_error_string	ErrorString Method	ErrorString Method
Error handling	Error Handling	Error Handling	Error Handling

EEPROM Memory

The ETH32 provides 256 bytes of EEPROM memory on the microcontroller which you can use for any data that you wish to store. The contents of the memory are preserved through power losses and reset, so it is perfect for storing device identification, calibration data, or other non-volatile data. EEPROM storage functionality is included with firmware v3.000 and greater.

Writing to EEPROM memory is a relatively slow process, which will temporarily disrupt event monitoring on the device. See the user manual for specific timing information.

Applicable Functions and Information

Task	C / C++	.NET Languages	Visual Basic 6
Read EEPROM Bytes	eth32_get_eeprom	GetEeprom Method	GetEeprom Method
Write EEPROM Bytes	eth32_set_eeprom	SetEeprom Method	SetEeprom Method

Other Functionality

The ETH32 provides a reset command that allows you to reset the port settings back to their poweron default values. For example, this sets all ports back to input mode and disables PWM channels, etc. The full list of settings that are affected is included in the description of the reset function. Note that doing a reset does not affect the network configuration of the device (its IP address, etc).

The ETH32 maintains a set of *connection flags* for each active connection to the device. Currently, the connection flags are used to indicate whether there was ever any data that the ETH32 had to discard for that connection due to lack of outgoing buffer space. These flags are informational only and in the majority of cases you do not need to consider them at all. More information is provided in the function reference, linked to below.

Applicable Functions

Task	C / C++	.NET Languages	Visual Basic 6
Reset	eth32_reset	ResetDevice Method	ResetDevice Method
Read/reset connection flags	eth32_connection_flags	ConnectionFlags Method	ConnectionFlags Method

Configuration and Detection

Starting with v2.00, the ETH32 API includes functionality that allows your applications to detect the presence of ETH32 devices on the local network, retrieve their current network configuration settings, and store new configuration settings into devices. One of the main reasons for including this functionality is the ability for ETH32 devices with firmware v3.000 and greater to use DHCP to automatically acquire an IP address. If the device is using DHCP, your application needs some way of finding out which IP address has been assigned to the device so your application can connect to it. If you control the configuration of the DHCP server, in most cases it is possible to configure the DHCP server to assign a specific, fixed IP address to a specific device based on its MAC address. In that case, your application could always connect to that fixed IP address and know that the ETH32 had that address. And of course, as before, the ETH32 device can still be configured with a static IP address, yielding a similar result. However, when DHCP is used and not configured to provide a fixed IP address, you will most likely want to utilize this functionality of the API to determine the IP address that has been assigned.

This functionality of the API is only able to detect and configure devices that are on the same local network segment as your PC. The functionality works by sending out UDP broadcast packets onto the network. This is the only practical way to detect the devices, since the specific address of each is unknown. However, network routers do not forward broadcast packets on to other network segments, so the API will not be able to detect any ETH32 devices that are past routers. Devices that are behind normal switches and hubs will be detected. Note that these restrictions only apply to the configuration and detection functionality. The normal ETH32 communications are implemented with standard (unicast) TCP/IP communications, and will work between any PC and device regardless of any routers in between, including over the Internet, as long as there is a valid route between them.

The API includes two different ways to detect devices. The Query function detects every ETH32 device on the local network, and returns all of the available settings and device information about each device. On the other hand, the Discover IP function allows you to discover just the active IP address and settings for a particular device that you specify by its MAC address or serial number. The Discover IP function is intended to be the standard method for your application to determine the IP to connect to when working with a device using DHCP. Your application should know the MAC address or the serial number of the device, and when provided to Discover IP, only information about that device will be returned. When a filter is provided to Discover IP, it is able to return immediately with that information as soon as the requested device replies. On the other hand, the Query functionality does not know how many replies will come in, so it delays a short period to allow replies to arrive, and also sends out the broadcast requests multiple times in case any network congestion causes interference. Query also sends two different types of packets out and potentially receives two different types of replies from each device, which is necessary to determine every available detail about the device, whereas Discover IP only sends and receives one type

of packet, which provides the active IP settings and the DHCP status of the device. So to summarize, since Query is attempting to find every bit of information about every device on the network, it will necessarily take longer than Discover IP, which is able to return as soon as the requested device replies. Note that if a filter is not provided to Discover IP, then it attempts to find all devices, and will take as long as Query. In that case, there is little difference except that Query sends out two different types of queries and Discover IP sends only one type. The packet type that Discover IP uses is supported on devices with firmware v3.000 and greater (those that support DHCP).

Depending on if and how you decide to detect devices, it may help to know how device serial numbers are constructed. An example of a serial number as it would be printed on the product label is 105-AB291. Each serial number is made up of three components: Product ID, Batch number, and Unit number. The Product ID is constant for all ETH32 devices, which is defined to be 105. As the example shows, this is printed first and followed by a dash. Batch and Unit numbers taken together are unique to each ETH32 device. The batch number is represented by letters in the serial number, such as the AB in the example. Internally and in the API code, the batch number is represented numerically, but when the serial number is printed, it is shown with letters. The letters are NOT hexadecimal, but rather in the style of Microsoft Excel column identifiers, although starting with AA. So, AA means 0, AB means 1, ..., AZ means 25, BA means 26, and so on. Lastly, the unit number is simply shown numerically and zero-padded out to three digits if necessary. The API includes a function to take the individual components and format a serial number string in the same way it is printed on the device.

The broadcast address to which packets are sent out may depend on your particular network configuration or PC, so it can therefore be specified to the API by your application. In most cases, a broadcast address of 255.255.255.255 is suitable.

The API also includes functions to convert between a string representation of an IP address, and the binary representation that is used by the Configuration / Detection functionality of the API.

Applicable Functions and Information

Task	C / C++	.NET Languages	Visual Basic 6
Query	eth32cfg_query	Query Method	Query Method
Discover IP	eth32cfg_discover_ip	DiscoverIp Method	DiscoverIp Method
Get results	eth32cfg_get_config	Result Property	Result Property
Free memory for results	eth32cfg_free	Free Method	Free Method
Store new configuration to device	eth32cfg_set_config	SetConfig Method	SetConfig Method
Broadcast address	Parameters to above functions	BroadcastAddress Property BroadcastAddressString Property	BroadcastAddress Property BroadcastAddressString Property
IP Address Conversion	eth32cfg_ip_to_string eth32cfg_string_to_ip	IpConvert Method IpConvertToString Method IpConvertToNetIPAddress Method	IpConvert Method IpConvertToString Method
Format a serial number string	eth32cfg_serialnum_string	SerialNumString Method	SerialNumString Method
MAC Address Conversion	None	MacConvert Method MacConvertToString Method	MacConvert Method MacConvertToString Method

Plugins

The Configuration / Detection functionality of the API supports loading in certain pre-defined plugins that can be used to help listen for responses from the devices and/or to help provide information about the PC's available network cards (interfaces) and their IP addresses. In most cases, using a plugin is not necessary.

There are currently three possible settings for the plugin available on Windows systems. This is not currently implemented for Linux systems.

Plugin options:

- None - No plugin used.
- System - The Windows API is used to provide information about the network interfaces on the PC. Using this plugin does not affect how queries are sent out or how responses are received.
- WinPcap - This plugin loads and utilizes the WinPcap library, if installed and available on the PC. This plugin is used to provide information about the network interfaces on the PC, as well as to "sniff" for the replies that ETH32 devices send in response to our queries. Using this plugin does not affect how queries are sent out.

Only one of the above plugins may be active at one time, and whichever plugin is active will apply to your entire application process (although it only affects the Configuration and Detection functionality). For example, if you load the WinPcap plugin and choose a network interface to sniff on, your application can still utilize the Discover IP functionality in the same way as always, but internally, the API will be using WinPcap to sniff for responses.

If your application has already loaded one plugin and is going to load another one instead, you must first make sure that you free any list of network interfaces that you have retrieved using the loaded plugin. Since plugins are loaded for the entire process and don't have a "handle," the API does not attempt to do this for you, and you must make sure you do it yourself.

Notes about WinPcap

In certain situations, a personal firewall (either Windows Firewall, or third party) on the PC may interfere with receiving responses back from ETH32 devices. Typically, this is only a problem when the ETH32 has not been properly configured with a valid IP for the network to which it is connected. While disabling the firewall is one solution, it may be desirable to use a "sniffer" to see the packets on the network before they get to the PC's firewall. WinPcap is a free and open-source sniffer library that provides this functionality. If WinPcap is installed on your PC, you can load this plugin in the ETH32 API to utilize the sniffing capability of WinPcap. There are a few items to be aware of:

- Depending on how WinPcap is installed, it may be necessary to run your application with Administrator rights in order for WinPcap to work properly.
- WinPcap needs to be told which specific network interface it should listen on. Therefore, it is necessary to use the ETH32 API functions to retrieve a list of the available network interfaces and choose one of them.
- The WinPcap plugin is not currently supported in the 64-bit version of the ETH32 API.

WinPcap downloads and information are at its web site: http://www.winpcap.org/

Applicable Functions and Information

Task	C / C++	.NET Languages	Visual Basic 6
Load Plugin	eth32cfg_plugin_load	Load Method	Load Method
Retrieve network interface information	eth32cfg_plugin_interface_list eth32cfg_plugin_interface_address eth32cfg_plugin_interface_type eth32cfg_plugin_interface_name	GetInterfaces Method NetworkInterface Property	GetInterfaces Method NetworkInterface Property
Choose network interface for sniffing	eth32cfg_plugin_choose_interface	ChooseInterface Method	ChooseInterface Method
Free list of network interfaces	eth32cfg_plugin_interface_list_free	Free Method	Free Method

Programming Languages

Libraries and support files to assist in using the ETH32 are provided for several languages. Please see the appropriate section for information regarding the programming language you will be using. Introductory information as well as a comprehensive reference is provided for each supported language.

The core of the ETH32 API is written in C and is contained in the eth32api.dll file on Windows or the libeth32*(version)*.so file on Linux. When the API is used from C or C++, your program directly calls functions from the core API. For other languages, a helper (or "wrapper") class is provided to make the use of the API easier. The wrapper class provides methods and properties which internally call the appropriate functions from the core API. Regardless, that explanation is simply provided for your information and it is not required that you understand how all of that fits together.

C/C++

Getting Started

There are just a few steps needed to use the ETH32 API from the C or C++ language. First you must copy the eth32.h header file from the API distribution files into your project's source code directory (unless you have already copied the header into your system's or compiler's header directory). There is a slightly different header file for Windows and Linux, so be sure to copy the correct one. Then, simply include the file at the top of any code module that uses functions from the API, as follows:

```
#include "eth32.h"
```

The only other special step is ensuring that your application is correctly linked with the ETH32 API library when your application is compiled. This varies depending on the compiler and the platform which you are using.

Microsoft Visual Studio C/C++ (Unmanaged)

These instructions apply to MS Visual Studio 4-6 as well as Visual Studio .NET when compiling unmanaged C++ applications (project type of Win32 Project). The term unmanaged means that the code is not using the .NET framework and garbage collection system. If you are using managed C++.NET, please see the instructions in the section for .NET languages.

In order for your application to link successfully, the compiler must be instructed to link with the eth32api.lib file. The purpose of the eth32api.lib file is to inform the compiler and linker of which functions are included in the eth32api.dll file. After your program is built (compiled and linked), your resulting application is not dependent on the eth32api.lib file, only on the eth32api.dll file.

For Visual Studio 4-6:

- Copy the eth32api.lib file into the same directory as your project source code.

- Open the project properties (depending on Visual Studio version, called Project Properties, Project Settings, or Build Settings)
- Under the Link tab, add eth32api.lib to the list of libraries in the Object/library modules list

For Visual Studio .NET Unmanaged (Win32) Projects:

- Copy the eth32api.lib file into the same directory as your project source code.
- Open the Project Properties (Right-click on the project in the Solution Explorer tree-view and select Properties).
- Navigate to the Configuration Properties -> Linker -> Input configuration page
- Under Additional Dependencies, enter eth32api.lib

Borland C/C++ Compilers

Borland C/C++ compilers also must have a .lib file in order to link successfully. However, the eth32api.lib file is in Microsoft (COFF) format and must be converted to the Borland format (OMF). A file is included in the API distribution that has already been converted to the OMF format and is named beth32api.lib

For your reference, the eth32api.lib file was converted to the beth32api.lib file with Borland's COFF2OMF utility using the command:

```
COFF2OMF -lib:ca eth32api.lib beth32api.lib
```

GNU C/C++ Compiler (GCC) on Linux

In order for your application to link successfully, the compiler must be instructed to link with the libeth32 library. Both shared and static versions of the library are provided in the distribution. We recommend using the shared library because it allows your applications to benefit from future updates of the library without recompiling each application. Note that before using the shared library, it must be installed in one of the system library directories (typically /usr/lib). This is already done if you have successfully run the Linux installation script.

Because the ETH32 API is multithreaded, you must compile your applications with the -pthread compiler flag. This is always necessary, even if your application doesn't directly create its own threads. Note that if you forget to include this flag, your application may work, but will exhibit strange behavior or crash in certain situations. If you ever notice this happening, please double check that you included the -pthread flag.

The compiler command for using the shared library will be similar to:

```
gcc -pthread myfile.c -leth32
```

When using the static library, copy the libeth32.a file into your source code directory and use a compiler command similar to:

```
gcc -pthread myfile.c libeth32.a
```

The shared library may also be used with the dlopen() and related functions, although most people will have no need to do so. If you do, though, please observe these points:

- Compile your application with the -pthread flag.
- Never dlclose() the library. There are a few situations where doing so can cause problems, so the best policy is to never dlclose() the library.

Error Handling

When using the API, errors may occur for a variety of reasons. For example, if the ETH32 device is powered off, there will be an error when trying to connect to it. As another example, if you try to read the value of a port, but specify a non-existent port number, an error will occur. All of the API functions have return values that indicate whether or not an error occurred.

The API functions return a value of zero when no error has occurred, or a negative value (one of the error codes listed below) when an error has occurred.

Note about eth32_open

The eth32_open function is different from the other functions. It returns the newly created handle to the device on success. The only invalid handle value is zero, which indicates an error if it is returned. If you need the actual error code, that can be returned through the parameter list.

Error Codes

The following error codes are defined in the header file:

- ETH_GENERAL_ERROR - A miscellaneous or uncategorized error has occurred.
- ETH_CLOSING - Function aborted because the device is being closed.
- ETH_NETWORK_ERROR - Network communications error. Connection was unable to be established or existing connection was broken.
- ETH_THREAD_ERROR - Internal error occurred in the threads and synchronization library.
- ETH_NOT_SUPPORTED - Function not supported by this device.
- ETH_PIPE_ERROR - Internal API error dealing with data pipes.
- ETH_RTHREAD_ERROR - Internal API error dealing with the "Reader thread."
- ETH_ETHREAD_ERROR - Internal API error dealing with the "Event thread."

- ETH_MALLOC_ERROR - Error dynamically allocating memory.
- ETH_WINDOWS_ERROR - Internal API error specific to the Microsoft Windows platform.
- ETH_WINSOCK_ERROR - Internal API error in dealing with the Microsoft Winsock library.
- ETH_NETWORK_INTR - Network read/write operation was interrupted.
- ETH_WRONG_MODE - Something is not configured correctly in order to allow this functionality.
- ETH_BCAST_OPT - Error setting the SO_BROADCAST option on a socket.
- ETH_REUSE_OPT - Error setting the SO_REUSEADDR option on a socket.
- ETH_CFG_NOACK - Warning - device did not acknowledge our attempt to store a new configuration.
- ETH_CFG_REJECT - Device has rejected the new configuration data we attempted to store. Configuration switch on device may be disabled.
- ETH_LOADLIB - Error loading an external DLL library.
- ETH_PLUGIN - General error with the currently configured plugin/sniffer library.
- ETH_BUFSIZE - A buffer provided was either invalid size or too small.
- ETH_INVALID_HANDLE - Invalid device handle was given.
- ETH_INVALID_PORT - The given port number does not exist on this device.
- ETH_INVALID_BIT - The given bit number does not exist on this port.
- ETH_INVALID_CHANNEL - The given channel number does not exist on this device.
- ETH_INVALID_POINTER - The pointer passed in to an API function was invalid.
- ETH_INVALID_OTHER - One of the parameters passed in to an API function was invalid.
- ETH_INVALID_VALUE - The given value is out of range for this I/O port, counter, etc.
- ETH_INVALID_IP - The IP address provided was invalid.
- ETH_INVALID_NETMASK - The subnet mask provided was invalid.
- ETH_INVALID_INDEX - Invalid index value.
- ETH_TIMEOUT - Operation timed out before it could be completed.

Structures

There are a few data structures defined by the API that are passed to or from functions. Each is described below.

eth32_event

The eth32_event structure holds all of the information about an event that has fired. It is passed from the API to your code when information about an event is retrieved, for example, with the eth32_dequeue_event function.

```
typedef struct
{
        int id;
        int type;
        int port;
        int bit;
        int prev_value;
        int value;
        int direction;
} eth32_event;
```

- id - The user-assigned event ID that you gave this event when enabling it.
- type - Event type, as defined by the constants EVENT_DIGITAL, EVENT_ANALOG, EVENT_COUNTER_ROLLOVER, EVENT_COUNTER_THRESHOLD, and EVENT_HEARTBEAT.
- port - For digital events, this specifies the port number the event occurred on. For analog events, it specifies the event bank number (0 or 1), and for counter events, it specifies which counter the event occurred on.
- bit - For a digital bit event, this specifies the bit number that changed. For an analog event, it specifies the analog channel, and for a digital port event, this will be -1.
- prev_value - The old value of the bit, port, or analog channel (as appropriate) before the event fired.
- value - The new value of the bit, port, or analog channel that caused the event to fire. In the case of counter events, this indicates the number of times the event occurred since the last time this event was fired (almost always 1).
- direction - Indicates whether the new value of the bit, port, or channel is greater or less than the previous value. It is 1 for greater than or -1 for less than.

eth32_handler

The eth32_handler structure is used to specify how events should be handled when they occur.

```
typedef struct
{
        int type;
        int maxqueue;
        int fullqueue;
        eth32_eventfn eventfn;
        void *extra;
        HWND window;
        unsigned int msgid;
        WPARAM wparam;
        LPARAM lparam;
} eth32_handler;
```

- type - Specifies how events should be handled and received by your code. It can be any of the following:
    - HANDLER_NONE - Your code will not be notified of new events. However, you may still receive event information by using the event queue functions.
    - HANDLER_CALLBACK - A callback function written and specified by you will be called whenever an event occurs. All of the data pertaining to the event will be passed to your callback function.
    - HANDLER_MESSAGE - For Windows platforms only. A Windows message that you specify will be sent to the window that you specify whenever an event occurs. No event data is included with the windows message. Therefore, this option should be used along with the event queue functions in order to actually receive the event data. In other words, the Windows message indicates that you should check the event queue for new events.

  The best option is usually determined by how and where you wish to receive event information in your program's code. If you are using a programming language that does not handle multiple threads well, you should not use the callback handler, since your callback function is called from a separate thread.

- maxqueue - Applies only to HANDLER_CALLBACK. If a callback function takes a while to finish executing and more events are received during that time, they are queued up by the API. This specifies the maximum number of events that are allowed to be queued.

- fullqueue - Applies only to HANDLER_CALLBACK. If the queue is already full and more events arrive, this specifies what to do. QUEUE_DISCARD_NEW specifies that the newly arriving events will be discarded if the queue is full. QUEUE_DISCARD_OLD specifies that the oldest events in the queue should be discarded and shifted out to make room for the new events at the end of the queue.

- eventfn - Applies only to HANDLER_CALLBACK. Specifies the address of your callback function. See the Event Callback Function section for more information about the callback function.

- extra - Applies only to HANDLER_CALLBACK. Whatever value is specified here will be passed to the "extra" parameter of the callback function whenever it is called. It may be any value you choose.

- window - Applies only to HANDLER_MESSAGE. Specifies the handle of the window to which a message should be sent when an event occurs.
- msgid - Applies only to HANDLER_MESSAGE. Specifies the message ID that should be sent. For example, WM_COMMAND.
- wparam - Applies only to HANDLER_MESSAGE. Specifies the wparam message parameter that should be sent.
- lparam - Applies only to HANDLER_MESSAGE. Specifies the lparam message parameter that should be sent.

Main Function Reference

Details are given below about each function provided by the core ETH32 API.

eth32_close

```
int eth32_close(eth32 handle);
```

Summary

This function closes the connection to the ETH32 device and cleans up all of the resources within the API that were used for the connection. After this function returns, the handle should be considered invalid and should not be used again.

Parameters

- handle - The value returned by the eth32_open function.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

Remarks

You should be careful to always call this function when you are finished using the device. The device has a limited number of connections it can support and if you do not call eth32_close and your application continues executing, you will continue using one of those connections. If you fail to call eth32_close, any connections will be automatically closed by the operating system when your application terminates.

See Also

eth32_open, eth32_verify_connection

---

eth32_connection_flags

```
int eth32_connection_flags(eth32 handle, int reset, int *flags);
```

Summary

The ETH32 device maintains several flag bits for each individual active TCP/IP connection. The flags indicate conditions that are (or were) present for that connection. Currently, these flags are used to indicate whether any data that needed to be sent to your application from the ETH32 device had to be discarded due to lack of queue space. This function retrieves the flags for this connection to the device. If instructed to do so, the function also clears all of the flags for this connection to zero immediately after retrieving them.

Parameters

- handle - The value returned by the eth32_open function.
- reset - If nonzero, specifies that the flags for this connection should be reset to zero immediately after retrieving them.
- flags - Pointer to a variable which will receive the current flags value for this connection. See the remarks below for a list of the individual flags that can make up this value.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

Remarks

To understand the role of the connection flags, consider the following example. Suppose that digital events are enabled on port 0, bit 0 for your connection to the ETH32. Now suppose that port 0, bit 0 begins pulsing rapidly, generating a steady stream of event notifications. Finally, suppose that the connection to your application is having trouble (losing packets, etc). Due to the nature of TCP/IP, the event notifications must be retained in the queue of the ETH32 device until a TCP/IP acknowledgement for them has been received from the PC (in case they need to be retransmitted). If the TCP/IP acknowledgements do not come promptly and the events keep occurring, the queue will eventually fill up and the ETH32 device will be forced to simply discard any new event notifications. Although this scenario is undesirable and should be avoided, if it does happen, it is helpful for your application to be able to detect that it happened and that data was lost. The flags keep track of this individually for each TCP/IP connection (that is, a full queue on one connection will not affect flags on another). Note that the flags are informational only - they do not affect the behavior of the device.

The flags value can be made up of a bitwise OR of any or all of the following individual flags. Each flag indicates which kind of data had to be discarded due to a full queue.

- CONN_FLAG_RESPONSE - Response to a query for information (for example eth32_input_byte).
- CONN_FLAG_DIGITAL_EVENT - Digital event notification.
- CONN_FLAG_ANALOG_EVENT - Analog event notification.
- CONN_FLAG_COUNTER_EVENT - Counter event (rollover or threshold) notification.

Once a flag is set, it will remain set until it is reset back to zero by passing a nonzero number to the *reset* parameter of this function. In this case, the flags will only be reset to zero if the connection has enough space to queue up the reply data. In other words, the flags will not be lost if the response itself is unable to be queued.

The connection flags for new connections always start out as zero. When the eth32_reset function is called, the flags for the connection it was received on are cleared, but the flags for any other active connections are not affected.

Example

```
eth32 handle;
int result;
int flags;

// .... Your code that establishes a connection here

// Retrieve the connection flags for this connection and
// simultaneously clear them to zero.
result=eth32_connection_flags(handle, 1, &flags);
if(result)
{
        // Handle error
}

// See which flags are set
if(flags & CONN_FLAG_RESPONSE)
{
        // The device ran out of queue space at some point
        // when it was trying to respond to a query for information.
}

if(flags & CONN_FLAG_DIGITAL_EVENT)
{
        // Some digital event data was lost due to running out
        // of queue space.
}

// and so on
```

See Also

eth32_verify_connection

---

eth32_dequeue_event

```
int eth32_dequeue_event(eth32 handle, eth32_event *event, int timeout);
```

Summary

This function retrieves information about an event from the internal API event queue and removes that entry from the queue. If the queue is empty and this function is instructed to do so, it will wait for an event to arrive from the device.

Events are dequeued in the sequence they arrived from the ETH32 device. The event queue must be enabled by setting a nonzero maximum queue size using the eth32_set_event_queue_config function.

Parameters

- handle - The value returned by the eth32_open function.
- event - Pointer to a structure which will receive all of the information about the event firing.
- timeout - If the queue is empty, specifies how long to wait for an event to arrive. A positive number instructs the function to wait up to that many milliseconds, zero specifies that it should not wait, but instead return ETH32_TIMEOUT immediately, and a negative value specifies the function should wait indefinitely for an event to arrive.

Return Value

This function returns zero if an event was successfully dequeued. If an error occurs or the function times out waiting for an event, a negative error code is returned. Please see the Error Codes section for possible error codes.

Example

```
eth32 handle;
int result;
eth32_event event_info;

// .... Your code that establishes a connection here

// Retrieve the next event from the queue.  If there are no events
// in the queue, wait up to 5 seconds for one to arrive.
result=eth32_dequeue_event(handle, &event_info, 5000);
if(result==ETH32_TIMEOUT)
{
        // No event was in the queue and none arrived within 5 seconds
}
else if(result)
{
        // Some other error occurred
}
```

```
else
{
        // An event was successfully dequeued
        printf("Event ID %d was dequeued.  Its new value is %d\n", event_info.id, event_info.value);
}
```

See Also

eth32_empty_event_queue, eth32_set_event_queue_config

---

eth32_disable_event

```
int eth32_disable_event(eth32 handle, int type, int port, int bit);
```

Summary

This function instructs the ETH32 device to stop sending event notifications for the specified event on this connection to the device. It performs the opposite task of eth32_enable_event.

Parameters

- handle - The value returned by the eth32_open function.
- type - The type of event to enable. The valid event types are:
  - EVENT_DIGITAL - Digital I/O event. This includes port events and bit events.
  - EVENT_ANALOG - Analog event based on thresholds defined with the eth32_set_analog_eventdef function.
  - EVENT_COUNTER_ROLLOVER - Counter rollover event, which occurs when the counter rolls over to zero.
  - EVENT_COUNTER_THRESHOLD - Counter threshold event, which occurs when the counter passes a threshold defined with eth32_set_counter_threshold.
  - EVENT_HEARTBEAT - Periodic event sent by the device to indicate the TCP/IP connection is still good.
- port - For digital events, specifies the port number, for analog events, specifies the bank number, and for either counter event, specifies the counter number.
- bit - For digital events, this should be -1 for port events or the bit number (0-7) for bit events. For analog events, this specifies the analog channel number (0-7).

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_enable_event

---

eth32_empty_event_queue

```
int eth32_empty_event_queue(eth32 handle);
```

Summary

This function empties the event queue within the API. Since the events are queued within the API, this function does not have an effect on the ETH32 device itself.

Parameters

- handle - The value returned by the eth32_open function.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_get_event_queue_status, eth32_set_event_queue_config

---

eth32_enable_event

```
int eth32_enable_event(eth32 handle, int type, int port, int bit, int id);
```

Summary

This function enables reception of the specified event on the current connection. The ETH32 device only sends event notifications to those connections that specifically request them, so this function requests notification for the specified event from the device, as well as internally assigns the event an ID number provided by you.

Parameters

- handle - The value returned by the eth32_open function.

- type - The type of event to enable. The valid event types are:
  - EVENT_DIGITAL - Digital I/O event. This includes port events and bit events.
  - EVENT_ANALOG - Analog event based on thresholds defined with the eth32_set_analog_eventdef function.
  - EVENT_COUNTER_ROLLOVER - Counter rollover event, which occurs when the counter rolls over to zero.
  - EVENT_COUNTER_THRESHOLD - Counter threshold event, which occurs when the counter passes a threshold defined with eth32_set_counter_threshold.
  - EVENT_HEARTBEAT - Periodic event sent by the device to indicate the TCP/IP connection is still good.
- port - For digital events, specifies the port number, for analog events, specifies the bank number, and for either counter event, specifies the counter number.
- bit - For digital events, this should be -1 for port events or the bit number (0-7) for bit events. For analog events, this specifies the analog channel number (0-7).
- id - You may specify any number to be associated with this event.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

Remarks

The *id* parameter allows you to assign any arbitrary number to this particular event. The ID you assign is included with the event information whenever this event fires. The idea is that you can identify a particular event with a single comparison rather than needing to inspect several pieces of data such as the event type, port number, and bit number. The ID number is completely arbitrary and multiple events may be given the same ID number if desired. The ID numbers are stored within the API and are not sent to the ETH32 device.

One other minor technicality is that the heartbeat event is permanently enabled on the ETH32 device itself for purposes of connection maintenance. Therefore, for the heartbeat event, this function simply enables the event within the API, meaning that when the event comes in, rather than being discarded it will be added to the event queue (if it is enabled) and handled by the configured event handler. The one small side-effect to this fact is that if you have enabled reception of the heartbeat event and another connection calls eth32_reset, you will continue to receive heartbeat events, whereas all other event types will have been disabled on the device itself. Note that if you call eth32_reset on your own connection, it automatically disables the heartbeat event within the API for your connection, so in that case it is not an issue.

Example

```
eth32 handle;
int result;
eth32_event event_info;


// .... Your code that establishes a connection here


// This example shows using the event queue to receive events ...
// you could use a callback function instead.
// Enable the event queue
result=eth32_set_event_queue_config(handle, 1000, QUEUE_DISCARD_NEW);
if(result)
{
        // handle error
}

// Enable an event that will fire whenever port 2, bit 5 changes state
// Assign it an arbitrary ID of 1000
result=eth32_enable_event(handle, EVENT_DIGITAL, 2, 5, 1000);
if(result)
{
        // handle error
}

// Enable the rollover event on Counter 0
// Assign it an arbitrary ID of 1001
result=eth32_enable_event(handle, EVENT_COUNTER_ROLLOVER, 0, 0, 1001);
if(result)
{
        // handle error
}


// Somewhere later in your code .... you want to process any events in the
// queue.  Do not wait at all for events - just process the ones already
// in the queue.  eth32_dequeue_event will return zero as long as there
// was an event in the queue.
while( (result=eth32_dequeue_event(handle, &event_info, 0)) == 0 )
{
        switch(event_info.id)
        {
                case 1000:
                        printf("Port 2 bit 5 has changed to have value %d\n",
                               event_info.value);
                        break;
                case 1001:
                        printf("Counter 0 has rolled over %d times "
                               "since we last received this event.\n",
                               event_info.value);
                        break;
        }
}
```

See Also

Event Callback Function, eth32_disable_event

---

eth32_error_string

```
const char * eth32_error_string(int errorcode);
```

Summary

This function translates a numeric error code into a string which briefly describes the error. This function returns a pointer to the string stored in static memory. You must use caution to not modify or overwrite the contents of the string buffer. No connection to an ETH32 device is necessary to call this function. Therefore, this function does not take a handle as a parameter.

Parameters

- errorcode - The numeric error code to translate into a string. Any error code returned by any API function may be passed for this parameter.

Return Value

This function returns a pointer to a string that describes the given error code. The string is stored in static memory.

Example

```
eth32 handle;
int result;
int value;

// .... Your code that establishes a connection here

// Attempt to read a port.  If it fails, print a
// brief description of the error.
result=eth32_input_byte(handle, 1, &value);
if(result)
{
        printf("Failed to read port 1.  The error was: %s\n", eth32_error_string(result));
}
```

See Also

Error Handling Section

---

eth32_get_analog_assignment

```
int eth32_get_analog_assignment(eth32 handle, int channel, int *source);
```

Summary

This function retrieves the current physical channel assignment for the specified logical channel. Please see the eth32_set_analog_assignment function for more information about logical channels and the physical channels that may be assigned to them.

Parameters

- handle - The value returned by the eth32_open function.
- channel - Specifies the logical channel (0-7).
- source - Pointer to a variable which will receive the code indicating which physical channel the logical channel is assigned to. This will be one of the ANALOG_ constants defined in the description of the eth32_set_analog_assignment function.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_set_analog_assignment

---

eth32_get_analog_eventdef

```
int eth32_get_analog_eventdef(eth32 handle, int bank, int channel, int *lomark, int *himark);
```

Summary

This function retrieves the low and high thresholds defined for the specified analog event bank and channel. Please see the eth32_set_analog_eventdef function for more information about the analog event definition and thresholds.

Parameters

- handle - The value returned by the eth32_open function.
- bank - Identifies which bank of analog events from which to retrieve information (0 or 1).
- channel - Identifies the analog channel (0-7).

- lomark - Pointer to a variable which will receive the low threshold (8-bit value) for the analog event.
- himark - Pointer to a variable which will receive the high threshold (8-bit value) for the analog event.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

Remarks

Note that this function does not return the default value that was specified when the thresholds were set. This is because the default value is only used during the moment that the thresholds are defined and is not applicable after that point.

See Also

eth32_enable_event, eth32_input_analog, eth32_set_analog_eventdef

---

eth32_get_analog_reference

```
int eth32_get_analog_reference(eth32 handle, int *reference);
```

Summary

This function retrieves the current analog voltage reference setting from the device. Please see the eth32_set_analog_reference function for further description of the voltage reference setting and its possible values.

Parameters

- handle - The value returned by the eth32_open function.
- reference - Pointer to a variable which will receive the voltage reference setting code (one of the REF_ constants)

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_get_analog_state, eth32_input_analog, eth32_set_analog_reference

---

eth32_get_analog_state

```
int eth32_get_analog_state(eth32 handle, int *state);
```

Summary

This function retrieves the status of the device's Analog to Digital Converter (ADC) to determine whether it is currently enabled or disabled.

Parameters

- handle - The value returned by the eth32_open function.
- state - Pointer to a variable which will receive the status of the ADC. This will be ADC_DISABLED (0) or ADC_ENABLED (1).

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

Example

```
eth32 handle;
int result;
int adc_state;

// .... Your code that establishes a connection here

result = eth32_get_analog_state(handle, &adc_state);
if(result)
{
        // Handle error
}

if(adc_state==ADC_ENABLED)
{
        // ADC is enabled
}
else
{
        // ADC is disabled
}
```

See Also

eth32_get_analog_reference, eth32_input_analog, eth32_set_analog_state

---

eth32_get_counter_rollover

```
int eth32_get_counter_rollover(eth32 handle, int counter, int *rollover);
```

Summary

This function retrieves the currently configured rollover point for the specified counter. Please see the eth32_set_counter_rollover function for more information about counter rollover thresholds.

Parameters

- handle - The value returned by the eth32_open function.
- counter - Specifies the counter number (0 or 1).
- rollover - Pointer to a variable which will receive the current rollover threshold for the specified counter.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_get_counter_state, eth32_set_counter_rollover

---

eth32_get_counter_state

```
int eth32_get_counter_state(eth32 handle, int counter, int *state);
```

Summary

This function retrieves the current state of the specified counter from the ETH32 device. See the eth32_set_counter_state function for more information about counter states.

Parameters

- handle - The value returned by the eth32_open function.
- state - Pointer to a variable which will receive the current counter state. This will be set to one of COUNTER_DISABLED, COUNTER_FALLING, or COUNTER_RISING.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_get_counter_rollover, eth32_get_counter_value, eth32_set_counter_state, eth32_set_counter_value

---

eth32_get_counter_threshold

```
int eth32_get_counter_threshold(eth32 handle, int counter, int *threshold);
```

Summary

This function retrieves the currently configured event threshold for the specified counter. Only counter 0 supports an event threshold on the ETH32 device. Please see the eth32_set_counter_threshold function for more information about the counter event threshold.

Parameters

- handle - The value returned by the eth32_open function.
- counter - Specifies the counter number. Must be 0.
- threshold - Pointer to a variable which will receive the currently configured event threshold on the counter.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_get_counter_state, eth32_get_counter_value, eth32_set_counter_threshold

---

eth32_get_counter_value

```
int eth32_get_counter_value(eth32 handle, int counter, int *value);
```

Summary

This function retrieves the current value of the specified counter. After you have enabled the counter with the eth32_set_counter_state function, the value of the counter indicates how many times the counter has been incremented by the external counter input.

Parameters

- handle - The value returned by the eth32_open function.

- counter - Specifies the counter number (0 or 1).
- value - Pointer to a variable which will receive the current value of the specified counter. For counter 0 (a 16-bit counter), this may range from 0-65535. For counter 1 (an 8-bit counter), this may range from 0-255.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_get_counter_rollover, eth32_get_counter_state, eth32_set_counter_value

---

eth32_get_direction

```
int eth32_get_direction(eth32 handle, int port, int *direction);
```

Summary

This function retrieves the current direction register for the specified digital I/O port. See the eth32_set_direction function for further description of the direction register.

Parameters

- handle - The value returned by the eth32_open function.
- port - The port number (0-5).
- direction - Pointer to a variable which will receive the contents of the direction register for the specified port.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_get_direction_bit, eth32_set_direction, eth32_set_direction_bit

---

eth32_get_direction_bit

```
int eth32_get_direction_bit(eth32 handle, int port, int bit, int *direction);
```

Summary

This function retrieves the value of a single bit of a port's direction register. It is provided simply for convenience, since it internally calls the eth32_get_direction function to determine the value of the specified bit.

Parameters

- handle - The value returned by the eth32_open function.
- port - Specifies the port number (0-5).
- bit - Specifies the bit number (0-7) within the port.
- direction - Pointer to a variable which will receive the value of the specified direction bit of the specified port.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_get_direction, eth32_set_direction, eth32_set_direction_bit

---

eth32_get_eeprom

```
int eth32_get_eeprom(eth32 handle, int address, int length, void *buffer);
```

Summary

This function retrieves data from the non-volatile EEPROM memory of the device.

Parameters

- handle - The value returned by the eth32_open function.
- address - The starting location from which data should be retrieved (0-255).
- length - The number of bytes to retrieve. Valid values for this parameter depend on what is provided for the address parameter. For example, with an address of 0, you may specify a length of all 256 bytes, but with an address of 255, length may only be 1.
- buffer - The buffer into which the data should be stored. This must be at least as long as the number of bytes you are requesting.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_set_eeprom

---

eth32_get_event_handler

```
int eth32_get_event_handler(eth32 handle, eth32_handler *handler);
```

Summary

This function retrieves information about the currently installed event handler mechanism for this connection. This is all information that is internal to the API, so this function does not need to retrieve any information from the ETH32 device.

Parameters

- handle - The value returned by the eth32_open function.
- handler - Pointer to an eth32_handler structure which will be filled in with the information about the current event handler.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

Example

```
eth32 handle;
int result;
eth32_handler evhandler;

// .... Your code that establishes a connection here

// Determine which type of event handler is installed
result=eth32_get_event_handler(handle, &evhandler);
if(result)
{
        // Handle error
}

switch(evhandler.type)
{
        case HANDLER_NONE:
                // No event handler is enabled
                break;
        case HANDLER_CALLBACK:
```

```
                // A callback function is set up as an event handler
                break;
        case HANDLER_MESSAGE:
                // A Windows Message event handler is enabled
                break;
}
```

See Also

Event Callback Function, eth32_enable_event, eth32_set_event_handler

---

eth32_get_event_queue_status

```
int eth32_get_event_queue_status(eth32 handle, int *maxsize, int *fullqueue, int *cursize);
```

Summary

This function retrieves the current configuration and status information about the event queue within the API. It allows you to find out the maximum queue size that is currently configured, the queue behavior when the queue becomes full, and the current number of events currently waiting in the queue. Since the queue and its configuration are within the API, this function does not retrieve any information from the ETH32 device. Please see the eth32_set_event_queue_config function for more information about the event queue configuration.

Parameters

- handle - The value returned by the eth32_open function.
- maxsize - Pointer to a variable which will receive the currently configured maximum size (maximum number of events) that can be held in the queue.
- fullqueue - Pointer to a variable which will receive a value indicating the queue behavior after the queue becomes full. This can be QUEUE_DISCARD_NEW or QUEUE_DISCARD_OLD.
- cursize - Pointer to a variable which will receive the current number of events that are in the queue.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_dequeue_event, eth32_enable_event, eth32_set_event_queue_config

---

eth32_get_firmware_release

```
int eth32_get_firmware_release(eth32 handle, int *major, int *minor);
```

Summary

This function retrieves the release number (version number) of the firmware on the device. The firmware version consists of a major number and minor number. When displayed as a string, it is typically formatted as major.minor with minor zero-padded to three digits if necessary. For example, for release 2.001, the major number is 2 and the minor number is 1.

Parameters

- handle - The value returned by the eth32_open function.
- major - Pointer to a variable which will receive the major number of the firmware version.
- minor - Pointer to a variable which will receive the minor number of the firmware version.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

Example

```
eth32 handle;
int result;
int major;
int minor;

// .... Your code that establishes a connection here

result=eth32_get_firmware_release(handle, &major, &minor);
if(result)
{
        // Handle error
}

printf("The device's firmware version is %d.%03d\n", major, minor);
```

See Also

eth32_get_serialnum

---

eth32_get_led

```
int eth32_get_led(eth32 handle, int led, int *value);
```

Summary

This function allows you to retrieve the status of the two LED's built into the ETH32 device.

Parameters

- handle - The value returned by the eth32_open function.
- led - Specifies which LED (0 or 1).
- value - Pointer to a variable which will receive the status of the specified LED. Zero means the LED is off and nonzero means it is on.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_set_led

---

eth32_get_product_id

```
int eth32_get_product_id(eth32 handle, int *prodid);
```

Summary

This function retrieves the product ID from the device, which identifies the type/model of the device.

Parameters

- handle - The value returned by the eth32_open function.
- prodid - Pointer to a variable which will receive the product ID.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_get_serialnum

---

eth32_get_pwm_base_period

```
int eth32_get_pwm_base_period(eth32 handle, int *period);
```

Summary

This function retrieves the currently configured base period for the PWM channels. See the eth32_set_pwm_base_period function for more information about the PWM base period.

Parameters

- handle - The value returned by the eth32_open function.
- period - Pointer to a variable which will receive the currently configured PWM base period, in terms of number of PWM clock counts.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_get_pwm_channel, eth32_get_pwm_clock_state, eth32_get_pwm_duty_period, eth32_set_pwm_base_period

---

eth32_get_pwm_channel

```
int eth32_get_pwm_channel(eth32 handle, int channel, int *state);
```

Summary

This function retrieves the current state of a PWM channel on the ETH32 device. Please see the eth32_set_pwm_channel function for more information about the possible channel states and their meanings.

Parameters

- handle - The value returned by the eth32_open function.
- channel - Specifies the PWM channel number (0 or 1).
- state - Pointer to a variable which will receive the state of the channel. This will be set to PWM_CHANNEL_DISABLED, PWM_CHANNEL_NORMAL, or PWM_CHANNEL_INVERTED

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_get_pwm_base_period, eth32_get_pwm_clock_state, eth32_get_pwm_duty_period, eth32_set_pwm_channel

---

eth32_get_pwm_clock_state

```
int eth32_get_pwm_clock_state(eth32 handle, int *state);
```

Summary

This function retrieves the current state (enabled or disabled) of the device's PWM clock.

Parameters

- handle - The value returned by the eth32_open function.
- state - Pointer to a variable which will receive the status of the PWM clock. This will be set to PWM_CLOCK_DISABLED or PWM_CLOCK_ENABLED.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_get_pwm_base_period, eth32_get_pwm_channel, eth32_get_pwm_duty_period, eth32_set_pwm_clock_state

---

eth32_get_pwm_duty_period

```
int eth32_get_pwm_duty_period(eth32 handle, int channel, int *period);
```

Summary

This function retrieves the current duty period for the specified PWM channel on the ETH32 device. Please see the eth32_set_pwm_duty_period function for more information about the duty period.

Parameters

- handle - The value returned by the eth32_open function.
- channel - Specifies the PWM channel number (0 or 1).
- period - Pointer to a variable which will receive the current duty period for the specified channel, in terms of PWM clock counts.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_get_pwm_base_period, eth32_get_pwm_channel, eth32_get_pwm_clock_state, eth32_set_pwm_duty_period

---

eth32_get_pwm_parameters

```
int eth32_get_pwm_parameters(eth32 handle, int channel, int *state, float *freq, float *duty);
```

Summary

This function is provided for your convenience in working with all of the various PWM settings. It internally calls several of the other API functions to determine the current state of the specified PWM channel and calculate its configuration in more familiar terms (hertz and percentage). This function calculates the frequency and duty cycle of the channel from the PWM base period and the channel's duty period.

Parameters

- handle - The value returned by the eth32_open function.
- channel - Specifies the PWM channel number (0 or 1).
- state - Pointer to a variable which will receive the current state of the PWM channel. This may be PWM_CHANNEL_DISABLED, PWM_CHANNEL_NORMAL, or PWM_CHANNEL_INVERTED.
- freq - Pointer to a variable which will receive the current frequency of the PWM channels in Hertz.
- duty - Pointer to a variable which will receive the duty cycle of the PWM channel. This may range from 0.00 to 1.00, representing the duty cycle as a percentage.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_set_pwm_parameters

---

eth32_get_serialnum

```
int eth32_get_serialnum(eth32 handle, int *batch, int *unit);
```

Summary

This function retrieves the serial number of the ETH32 device in numeric format. It retrieves the batch number and the unit number, which are the two components of each device's serial number. To retrieve the serial number in string format, as it is printed on the device, please see the eth32_get_serialnum_string function.

Parameters

- handle - The value returned by the eth32_open function.
- batch - Pointer to a variable which will receive the batch number portion of the serial number.
- unit - Pointer to a variable which will receive the unit number portion of the serial number.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_get_serialnum_string

---

eth32_get_serialnum_string

```
int eth32_get_serialnum_string(eth32 handle, char *serial, int bufsize);
```

Summary

This function retrieves the serial number of the ETH32 device in string format as it is printed on the device. To retrieve the components of the serial number in numeric format, see the eth32_get_serialnum function.

Parameters

- handle - The value returned by the eth32_open function.
- serial - Pointer to a string buffer that will be receive the serial number string. The function will add a null termination byte to the end of the string.
- bufsize - Specifies how long, in bytes, the buffer pointed to by the *serial* parameter is. If the function determines the buffer length is not long enough to hold the serial number, it does not write anything in to the buffer and returns an error (ETH_INVALID_OTHER).

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

Remarks

The serial number is made up of several components and arranged as follows:

```
(productid)-(batch)(unit)
```

where:

- productid is a number identifying the product type/model. This number is returned by the eth32_get_product_id function.
- batch is the batch number formatted as two letters. 1 becomes AA, 2 becomes AB, etc.
- unit is the unit number, zero padded to 3 digits if necessary.

Example

```
eth32 handle;
int result;
char serial[50];

// .... Your code that establishes a connection here

// Retrieve the serial number
result=eth32_get_serialnum_string(handle, serial, sizeof(serial));
if(result)
{
        // Handle error
}

printf("The device’s serial number is %s\n", serial);
```

See Also

eth32_get_serialnum

---

eth32_get_timeout

```
int eth32_get_timeout(eth32 handle, unsigned int *timeout);
```

Summary

This function is used to retrieve the internal API timeout used for any function that requires a response from the ETH32 device. See the eth32_set_timeout function for further description of the timeout setting.

Parameters

- handle - The handle value returned by the eth32_open function.
- timeout - Pointer to a variable which will receive the currently configured timeout value.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_set_timeout

---

eth32_input_analog

```
int eth32_input_analog(eth32 handle, int channel, int *value);
```

Summary

This function retrieves an analog reading from one of the analog channels on the device. The analog readings are only meaningful when the ADC has been enabled (see eth32_set_analog_state). The analog readings are 10-bit values. See below for further explanation of their meaning.

Parameters

- handle - The value returned by the eth32_open function.
- channel - Specifies the logical analog channel (0-7) to read. Note that each logical analog channel may be arbitrarily assigned to physical channels using eth32_set_analog_assignment.
- value - Pointer to a variable which will receive the reading from the specified channel.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

Remarks

The reading that is obtained with this function is a 10-bit value (range of 0-1023) representing the voltage level relative to the analog reference voltage. The exact interpretation depends on whether a single-ended or differential channel has been selected (see eth32_set_analog_assignment).

For single-ended channels, the reading is:

```
(analog reading) = (channel voltage * 1024) / (voltage reference)
```

For example, a reading of 0 means 0V and a reading of 1023 means a voltage just under the voltage reference (assuming internal 5V reference, about 4.99V). Once you have the analog reading, you can calculate the input voltage that produced it by calculating:

```
voltage = (analog reading)/1024 * (voltage reference)
```

For differential channels, the reading is:

```
(analog reading) = 512 + (positive side voltage - negative side voltage) * GAIN * 512 / (voltage reference)
```

For example, assuming a gain of 1X, a reading of 0 means the positive pin is (voltage reference) volts less than the negative pin, a reading of 512 means the positive pin and negative pin are at the same voltage, and a reading of 1023 means the positive pin is almost (voltage reference) volts higher than the negative pin. Once you have the analog reading, you can calculate the voltage of the positive pin relative to the negative pin by calculating:

```
voltage = (analog reading - 512) / 512 * (voltage reference)
```

Example

```
// NOTE: Error handling omitted for clarity
eth32 handle;
int result;
int chan0;
float voltage;

// .... Your code that establishes a connection here

// Enable the Analog to Digital Converter
eth32_set_analog_state(handle, ADC_ENABLED);

// Configure logical channel 0 to read the physical channel 0 relative to ground (single-ended)
// This is the power-on default anyway, but is shown here for completeness:
eth32_set_analog_assignment(handle, 0, ANALOG_SE0);

// Configure the analog voltage reference to be the internal 5V source
eth32_set_analog_reference(handle, REF_INTERNAL);

// Finally, read the voltage on channel 0
eth32_input_analog(handle, 0, &chan0);
```

```
// Now, determine whether the voltage was >= 3V.  Remember
// we're using a 5V voltage reference.
if( chan0 >= (3.0/5.0 * 1024) )
{
        // The voltage on channel 0 was at least 3V
}
else
{
        // The voltage was less than 3V
}

// If you want to calculate the voltage:
voltage = chan0 / 1024.0 * 5.0;
```

See Also

eth32_set_analog_assignment, eth32_set_analog_reference, eth32_set_analog_state

---

eth32_input_bit

```
int eth32_input_bit(eth32 handle, int port, int bit, int *value);
```

Summary

This function retrieves the value of a single bit within a port. It is provided simply for convenience, since it internally calls the eth32_input_byte function to determine the value of the specified bit.

Parameters

- handle - The value returned by the eth32_open function.
- port - Specifies the port number (0-5) to read.
- bit - Specifies the bit number (0-7) of the port to read.
- value - Pointer to a variable which will receive the current value (0 or 1) of the specified bit.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_input_byte, eth32_output_bit, eth32_set_direction_bit

---

eth32_input_byte

```
int eth32_input_byte(eth32 handle, int port, int *value);
```

Summary

This function retrieves the current input value of a specified port on the device. When a port is configured as an input port (using the eth32_set_direction function), the input value represents the voltage levels on the port's pins. For each bit, a low voltage (close to 0V) yields a 0-bit in the input value and a high voltage (close to 5V) yields a 1-bit.

Parameters

- handle - The value returned by the eth32_open function.
- port - Specifies the port number (0-5) to read.
- value - Pointer to a variable which will receive the current input value of the specified port.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

Example

```
eth32 handle;
int result;
int value;

// .... Your code that establishes a connection here

// Read the input value of port 2
result = eth32_input_byte(handle, 2, &value);
if(result)
{
        // Handle error
}

// See whether any of bits 0-3 are high (1)
if( (value & 0x0F) )
{
        // At least one of bits 0-3 are high
}
else
{
        // None of bits 0-3 are high
}
```

See Also

eth32_input_bit, eth32_output_byte, eth32_set_direction

---

eth32_input_successive

```
int eth32_input_successive(eth32 handle, int port, int max, int *value, int *status);
```

Summary

This function instructs the ETH32 device to read the specified port multiple times in succession until two consecutive reads yield the same result. This function is useful for situations where a multi-bit value is being read, for example, the output of a digital counter chip. When reading such a value, it is always possible to read the value during a transition state as bits are changing and an invalid value is represented. By requiring that two successive reads match, any invalid transition values are automatically ignored. The device continues to read the port until one of the following conditions is met:

- Two successive reads give the same port value. This value is returned.
- The port was read the maximum number of times specified in the command without a match occurring.

This functionality is implemented directly within the ETH32 device (as opposed to the API), making it very fast and efficient with network traffic.

Parameters

- handle - The value returned by the eth32_open function.
- port - Specifies the port number (0-3) to read.
- max - The maximum number of times to read the port (2-255).
- value - Pointer to a variable which will receive the port value. This will be the last value read from the port, regardless of whether or not two successive reads ever matched.
- status - Pointer to a variable which will receive the number of times the port had to be read to get a successive match. If no match was ever seen, this will be zero.

Return Value

This function returns zero on success and a negative error code on failure. Please note that the function is considered to succeed even if a matching value between two successive reads is never seen. Please see the Error Codes section for possible error codes.

Example

```
eth32 handle;
int result;
int value;
int status;

// .... Your code that establishes a connection here

// Read the value of port 0, limit to 20 reads
result=eth32_input_successive(handle, 0, 20, &value, &status);
if(result)
{
        // handle error
}

if(status==0)
{
        // Never saw the same value twice in a row
}
else
{
        printf("The port value is %d.\n", value);
}
```

See Also

eth32_input_byte, eth32_set_direction

---

eth32_open

```
eth32 eth32_open(char *address, WORD port, unsigned int timeout, int *result);
```

Summary

The eth32_open function is used to open a new connection to an ETH32 device. It returns a handle which you must save and pass to any other API functions you call. Note that your application may have connections open to several ETH32 devices at once, so the handle serves to identify each connection. This function does not reset the device or change its configuration in any way.

Parameters

- address - The IP address or DNS name of the ETH32 device.
- port - The TCP port to connect to. The ETH32 listens on TCP port 7152. The constant ETH32_PORT may be used here.
- timeout - Specifies the maximum time, in milliseconds, that the connection attempt may take, excluding resolving DNS. You may specify a timeout of zero to use the default timeout from the system's TCP/IP stack. Note that the function may time out in less time than you specify if the system's timeout is shorter.

- result - Receives the result/error code of the function. You may specify NULL if you are not interested in the error code. On a successful connection, a value of zero is stored to this parameter. On error, the error code is stored. See the Error Codes section for possible error codes.

Return Value

This function returns a handle to the device if it was successfully opened. You must save the handle value and pass it as a parameter to any other API functions that you call. The only return value that indicates an error is zero. If you want to receive the actual error code in the event of an error, use the *result* parameter specified above.

Example

```
eth32 handle;
int result;

handle=eth32_open("192.168.1.100", ETH32_PORT, 0, &result);
if(handle==0)
{
        printf("Error connecting to ETH32: %s\n", eth32_error_string(result));
        // handle error as appropriate in your code, prevent falling through
        // to code below.
}
// Now that we're connected, turn on an LED:
eth32_set_led(handle, 0, 1);
```

See Also

eth32_close, eth32_verify_connection

---

eth32_output_bit

```
int eth32_output_bit(eth32 handle, int port, int bit, int value);
```

Summary

This function alters a single bit of the output value of any I/O port without affecting the value of any other bits. See the eth32_output_byte function for further description of the output value.

Parameters

- handle - The value returned by the eth32_open function.
- port - The port number (0-5).
- bit - The bit number (0-7).
- value - Any nonzero number sets the bit to 1 and zero clears the bit to 0.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

Remarks

This function alters the specified bit's value in a single operation directly on the ETH32 device. In other words, it does NOT read the current value over the network, modify it and then write it back. By doing it in a single operation, this avoids the potential of inadvertently overwriting changes made to other bits by other connections.

Port 3 shares its pins with the analog channels. When the ADC is enabled, all pins of port 3 are forced into input mode and the output value is set to zero. Port 3's output value cannot be modified while the ADC is enabled.

See Also

eth32_input_bit, eth32_output_byte, eth32_set_direction_bit

---

eth32_output_byte

```
int eth32_output_byte(eth32 handle, int port, int value);
```

Summary

This function writes a new output value to one of the digital I/O ports on the device. When the port is configured as an output port (using the eth32_set_direction function), each bit of the output value determines the voltage (0 or 5V) of the corresponding bit of the port. When the port is configured as an input port, any 1-bits in the output value enables a weak pullup for that bit of the port.

Parameters

- handle - The value returned by the eth32_open function.
- port - The port number to write to (0-5).
- value - The new value for the port. This may be 0-255 for ports 0-3 and 0-1 for the single-bit ports 4 and 5.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

Remarks

Port 3 shares its pins with the analog channels. When the ADC is enabled, all pins of port 3 are forced into input mode and the output value is set to zero. Port 3's output value cannot be modified while the ADC is enabled.

Example

```
eth32 handle;
int result;

// .... Your code that establishes a connection here

// Set port 0 pins to be outputs
result=eth32_set_direction(handle, 0, 255);
if(result)
{
        // handle error here
}

// Write a new value for port 0
result=eth32_output_byte(handle, 0, 85);
if(result)
{
        // handle error here
}
```

See Also

eth32_input_byte, eth32_output_bit, eth32_readback, eth32_set_direction

---

eth32_pulse_bit

```
int eth32_pulse_bit(eth32 handle, int port, int bit, int edge, int count);
```

Summary

This function outputs a burst of pulses on the port and bit specified. This can be useful, for example, in quickly clocking an external digital counter a specified number of times. You should ensure that the specified bit is configured as an output bit before calling this function. The ETH32 device implements the pulse functionality (as opposed to the API), which means it is performed very quickly and is efficient for network traffic.

Parameters

- handle - The value returned by the eth32_open function.
- port - The port number (0-5).

- bit - The bit number (0-7) on the specified port that should be pulsed.
- edge - Specifies whether the pulses should be falling or rising edge. This parameter can accept either of these constants, which define a single pulse as follows:
  - PULSE_FALLING - Bit is set low, then high, for each pulse.
  - PULSE_RISING - Bit is set high, then low, for each pulse.
- count - The number of times to pulse the bit. May be up to 255.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

Remarks

The falling edge mode would typically be used on a bit that is initially high (and likewise rising edge with low), but this is not required. If a single falling edge pulse is performed on a bit that is already low, the pulse will end up simply setting the bit high. The reverse applies to a rising edge pulse where the bit is already high.

See Also

eth32_output_bit, eth32_set_direction_bit

---

eth32_readback

```
int eth32_readback(eth32 handle, int port, int *value);
```

Summary

This function retrieves (reads back) the current output value for the specified port. This is the value that was last written by calling eth32_output_byte or one or more calls to eth32_output_bit.

Parameters

- handle - The value returned by the eth32_open function.
- port - The port number to read back (0-5)
- value - Pointer to a variable which will receive the port's output value.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_output_bit, eth32_output_byte

---

eth32_reset

```
int eth32_reset(eth32 handle);
```

Summary

This function resets most aspects of the device to their power-up default status. It does not perform a "cold reset" of the device. All TCP/IP connections to the device are preserved and do not need to be reestablished. See the remarks below for a list of everything that is affected.

Parameters

- handle - The value returned by the eth32_open function.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

Remarks

The following parts of the device are reset by this function:

- All digital I/O ports are configured as inputs.
- The output values of all digital I/O ports are set to zero.
- The LED’s are turned off
- The Analog to Digital Converter is disabled.
- The analog voltage reference is configured to the external reference (REF_EXTERNAL).
- The analog channel assignments are all set to the single-ended channels. Logical channel 0 is set to single-ended channel 0, Logical channel 1 to single-ended 1, and so on.
- All events are disabled for all connections.

- Analog event definitions are cleared.
- Both counters are disabled.
- Counter values are set to zero.
- Counter rollover points are set to their highest possible values (0xFFFF for 16-bit counter 0, 0xFF for 8-bit counter 1).
- Counter event threshold (applies only to counter 0) set to zero.
- PWM channels are disabled and the main PWM clock is stopped.
- The PWM base period is set to its highest (lowest frequency) setting of 0xFFFF counts.
- The duty period of both PWM channels is set to zero.
- The connection flags are reset only for the connection that performed the reset. The connection flags for any other connections are not affected.

---

eth32_set_analog_assignment

```
int eth32_set_analog_assignment(eth32 handle, int channel, int source);
```

Summary

This function assigns a logical analog channel to one of the physical channels. The logical channel assignment specifies which physical pins are used to determine the value of the analog reading when that logical channel is read or monitored for events. There are eight logical channels, each of which may be arbitrarily assigned to physical channels using this function.

Parameters

- handle - The value returned by the eth32_open function.
- channel - The logical channel number (0-7) to configure.
- source - The code identifying which physical channel to assign to the specified logical channel. This may be any of the ANALOG_ constants defined in the tables below.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

Remarks

The logical channels simply provide a way to select which of the many physical channel sources listed below will be continually updated for reading on the device and, if configured to do so, monitored for analog events.

The assignments given to the logical channels may be completely arbitrary. Also, it is permissible to have more than one logical analog channel assigned to the same physical channel source. This can occasionally be advantageous for event monitoring. Since there are two possible event definitions per logical channel, assigning more than one logical channel to the same physical channel allows more than two event definitions on that physical channel.

When the device is first powered up or the eth32_reset function is called, the logical channel assignments revert to their defaults. Logical channel 0 is assigned to single-ended channel 0, logical channel 1 to single-ended channel 1 and so on.

The assignments made with this function are effective until they are either overwritten by calling the function again or the board is reset (hard reset or by calling eth32_reset). There is no limitation on how often you may reassign logical channels.

The following settings are the valid physical channel sources to which a logical channel may be assigned. The constant definition should typically be used in your source code, but its hexadecimal value is shown for reference.

For single-ended channels, the reading comes from the voltage of the specified pin with respect to ground.

Table 1. Single-Ended Channels

Constant	Value	Physical Pin
ANALOG_SE0	0x00	Port 3, Bit 0
ANALOG_SE1	0x01	Port 3, Bit 1
ANALOG_SE2	0x02	Port 3, Bit 2
ANALOG_SE3	0x03	Port 3, Bit 3
ANALOG_SE4	0x04	Port 3, Bit 4
ANALOG_SE5	0x05	Port 3, Bit 5
ANALOG_SE6	0x06	Port 3, Bit 6
ANALOG_SE7	0x07	Port 3, Bit 7

For differential channels, the reading comes from the voltage difference between two pins. It is permissible for either to be positive or negative with respect to the other. They are simply labeled positive and negative inputs to specify how the reading is determined. Please note that the voltage on each pin must still remain within the range of 0 to 5V with respect to the ground of the device.

Table 2. Differential Channels

Constant	Value	Positive Input	Negative Input	Gain
ANALOG_DI00X10	0x08	Port 3, Bit 0	Port 3, Bit 0	10x
ANALOG_DI10X10	0x09	Port 3, Bit 1	Port 3, Bit 0	10x
ANALOG_DI00X200	0x0A	Port 3, Bit 0	Port 3, Bit 0	200x
ANALOG_DI10X200	0x0B	Port 3, Bit 1	Port 3, Bit 0	200x
ANALOG_DI22X10	0x0C	Port 3, Bit 2	Port 3, Bit 2	10x
ANALOG_DI32X10	0x0D	Port 3, Bit 3	Port 3, Bit 2	10x
ANALOG_DI22X200	0x0E	Port 3, Bit 2	Port 3, Bit 2	200x
ANALOG_DI32X200	0x0F	Port 3, Bit 3	Port 3, Bit 2	200x
ANALOG_DI01X1	0x10	Port 3, Bit 0	Port 3, Bit 1	1x
ANALOG_DI11X1	0x11	Port 3, Bit 1	Port 3, Bit 1	1x
ANALOG_DI21X1	0x12	Port 3, Bit 2	Port 3, Bit 1	1x
ANALOG_DI31X1	0x13	Port 3, Bit 3	Port 3, Bit 1	1x
ANALOG_DI41X1	0x14	Port 3, Bit 4	Port 3, Bit 1	1x
ANALOG_DI51X1	0x15	Port 3, Bit 5	Port 3, Bit 1	1x
ANALOG_DI61X1	0x16	Port 3, Bit 6	Port 3, Bit 1	1x
ANALOG_DI71X1	0x17	Port 3, Bit 7	Port 3, Bit 1	1x
ANALOG_DI02X1	0x18	Port 3, Bit 0	Port 3, Bit 2	1x
ANALOG_DI12X1	0x19	Port 3, Bit 1	Port 3, Bit 2	1x
ANALOG_DI22X1	0x1A	Port 3, Bit 2	Port 3, Bit 2	1x
ANALOG_DI32X1	0x1B	Port 3, Bit 3	Port 3, Bit 2	1x
ANALOG_DI42X1	0x1C	Port 3, Bit 4	Port 3, Bit 2	1x
ANALOG_DI52X1	0x1D	Port 3, Bit 5	Port 3, Bit 2	1x

Note that the entries above which show both the positive side and negative side with the same input pin can be used for calibration of the differential amplifier. Any nonzero reading from those indicates an offset error within the differential amplifier which you can subtract out of other channels that share the same negative input and gain.

Table 3. Calibration Channels

Constant	Value	Description
ANALOG_122V	0x1E	Internal 1.22V Voltage Reference
ANALOG_0V	0x1F	0V (Ground)

The above two entries connect a logical channel to internal chip voltages. They can be used as calibration points to determine errors within the analog conversions.

See Also

eth32_get_analog_assignment

---

eth32_set_analog_eventdef

```
int eth32_set_analog_eventdef(eth32 handle, int bank, int channel, int lomark, int himark, int defaultval);
```

Summary

This function defines the event thresholds for a single logical analog channel in the specified analog event bank. The thresholds that are defined determine what analog readings will cause the event to fire. The thresholds allow the event logic on the ETH32 device to assign a current state (high or low) to the event. The event will be considered high if the analog reading is at or above the given hi-mark and will be considered low if at or below the given lo-mark. Whenever the state of the event changes (low to high or high to low), an event notification will be fired. When the analog reading is between the lo-mark and hi-mark, it will retain its previous value. This allows "hysteresis" to be built into the event so that a fluctuating signal will not cause an event to continuously fire. The thresholds are specified in 8-bit resolution, and thus they will be compared with the eight Most Significant Bits of the analog readings to determine when an event should be fired. The given hi-mark must be greater than the lo-mark.

Normally, the "initial state" (high or low) of the analog event definition is determined by the current level of the analog reading at the time the event definition is defined. However, if the current analog reading is between the lo-mark and hi-mark, an initial state cannot be accurately assigned. To deal with this, this function accepts a parameter that defines a default state to be used when the initial state cannot be determined. In all other situations (when the reading at the time of event definition is <= lo-mark or >= hi-mark) this parameter will simply be ignored.

Parameters

- handle - The value returned by the eth32_open function.
- bank - Specifies the event bank (0 or 1).
- channel - Specifies the logical channel (0-7).

- lomark - Low threshold, 8 Most Significant Bits (0-255).
- himark - High threshold, 8 Most Significant Bits (0-255).
- defaultval - Specifies whether the event should be considered high or low if the current analog reading is between lomark and himark. ANEVT_DEFAULT_LOW (0) specifies it should be considered to be low and ANEVT_DEFAULT_HIGH (1) specifies it should be considered high.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

Remarks

Please note that defining the thresholds with this function does not enable the current connection to actually receive the event notifications when they occur. These must be enabled using the eth32_enable_event function. Also note that the analog event thresholds are common to all connections. Changing the thresholds will affect other connections if they are utilizing that particular event.

Because the ETH32 device has two analog event banks, two events can be defined for each logical analog channel on the board. Applications can utilize both event banks independently to implement a number of different event notification schemes.

Example

```
// NOTE: Error handling omitted for clarity
eth32 handle;
int result;
int lomark;
int himark;

// .... Your code that establishes a connection here

// .... Your code that configures an appropriate event handler goes here (or later)

// Enable the Analog to Digital Converter
eth32_set_analog_state(handle, ADC_ENABLED);

// Configure logical channel 7 to read the physical channel 7 relative to ground (single-ended)
// This is the power-on default anyway, but is shown here for completeness:
eth32_set_analog_assignment(handle, 7, ANALOG_SE7);

// Configure the analog voltage reference to be the internal 5V source
eth32_set_analog_reference(handle, REF_INTERNAL);


// Define an event that fires when channel 7 goes above 3.5V or
// falls below 3.0V.  Remember that the thresholds must be calculated
// knowing the voltage reference (in this case 5V).  They also must be
// converted to the 8 Most Significant Bits from 10-bit by dividing by 4.
// If the current reading happens to be between the low and high threshold,
// we will default to the event starting out low.
lomark=3.0 / 5.0 * 1024 / 4;
himark=3.5 / 5.0 * 1024 / 4;
eth32_set_analog_eventdef(handle, 0, 7, lomark, himark, ANEVT_DEFAULT_LOW);
```

```
// Now that an event is defined in bank 0, channel 7, enable receiving
// events from it.
// We'll give this event an arbitrary ID of 8000
eth32_enable_event(handle, EVENT_ANALOG, 0, 7, 8000);

// You will now receive events through whatever event handler mechanism you
// have configured when channel 7 crosses the threshold to being over 3.5V or
// crosses to under 3.0V.
```

See Also

eth32_enable_event, eth32_get_analog_eventdef, eth32_input_analog

---

eth32_set_analog_reference

```
int eth32_set_analog_reference(eth32 handle, int reference);
```

Summary

This function instructs the Analog to Digital Converter to select the specified source as the reference voltage for conversions. The reference voltage determines the voltage level that will give the highest possible analog reading value. There are three possible voltages that may be used: An externally-generated voltage supplied on the analog reference pin, internal 5V, and internally generated 2.56V.

Parameters

- handle - The value returned by the eth32_open function.
- reference - A code indicating which voltage source to select. This may be the constants REF_EXTERNAL, REF_INTERNAL (internal 5V), or REF_256 (internal 2.56V)

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

Remarks

Note that whatever voltage source is selected will be internally connected to the external voltage reference pin. So for example, if you have a 4V source connected to the external reference pin, you should NOT configure the reference for REF_INTERNAL or REF_256 until you have disconnected the external reference pin.

Also note that if you connect a voltage to the external reference pin, it must not exceed 5V or go below 0V.

See Also

eth32_get_analog_reference, eth32_input_analog, eth32_set_analog_state

---

eth32_set_analog_state

```
int eth32_set_analog_state(eth32 handle, int state);
```

Summary

This function enables or disables the Analog to Digital Converter (ADC) portion of the ETH32 device. The ADC must first be enabled before any valid analog readings can be taken obtained.

Parameters

- handle - The value returned by the eth32_open function.
- state - Whether to enable (1) or disable (0) the ADC. The constants ADC_ENABLED and ADC_DISABLED may be used for this parameter.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

Remarks

Because the analog channels use the same physical pins as digital I/O port 3, enabling the ADC forces port 3 into input mode and sets the output value of port 3 to zero. Changes to the direction register or output value of port 3 are disabled while the ADC remains enabled. Note that regardless of what port 3's direction register and output value were at the time the ADC was enabled, if the ADC is later disabled, port 3 will be left in input mode with an output value of zero.

See Also

eth32_get_analog_state, eth32_input_analog, eth32_set_analog_reference

---

eth32_set_counter_rollover

```
int eth32_set_counter_rollover(eth32 handle, int counter, int rollover);
```

Summary

This function defines the maximum permissible value for a counter. After the counter reaches the given value, the next count will cause the counter to be reset to 0 and a rollover event notification will be sent to any connections that have enabled that rollover event. For example, with a rollover threshold set to 35, the counter value will progress as follows: ..., 33, 34, 35, 0, 1, ... Because the comparisons and reset are done directly in hardware, no counts are missed during a rollover.

The valid range of the rollover threshold is from 0 to the maximum value of the counter (65535 for 16-bit counter 0, and 255 for 8-bit counter 1). The powerup default rollover threshold is 255 for 8-bit and 65535 for 16-bit counters.

Parameters

- handle - The value returned by the eth32_open function.
- counter - Specifies the counter number (0 or 1).
- rollover - Specifies the rollover point for the counter. This may be 0-65535 for counter 0, and 0-255 for counter 1.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

Remarks

There is one special case involving rollover thresholds. When the counter value is manually set to exactly the threshold value using the eth32_set_counter_value function, the rollover will NOT occur and the rollover event will NOT fire on the next counter increment. Instead, the counter will increment past the threshold value. The event will not fire until the counter value has wrapped around and again exceeds the threshold. For example, suppose the rollover threshold is set to 10 on an 8-bit counter and the eth32_set_counter_value function is used to set the counter value to 10. As the input line pulses, the counter value would increment as follows: 11, 12, ..., 255, 0, 1, ..., 10, 0, 1, ..., 10, 0, ...

Please note that defining a rollover threshold with this function does not enable the current connection to actually receive the rollover event notifications when they occur. These must be enabled separately using the eth32_enable_event function. Also note that rollover thresholds are common to all connections. Changing the thresholds will affect other connections if they are utilizing that particular counter.

See Also

eth32_get_counter_rollover, eth32_set_counter_state

---

eth32_set_counter_state

```
int eth32_set_counter_state(eth32 handle, int counter, int state);
```

Summary

This function enables and disables the counters of the ETH32 device and configures which input signal edge (rising or falling) will increment the counter value. This function does not affect the current counter value in any way. In other words, a counter that is disabled and then enabled again will retain its value.

Parameters

- handle - The value returned by the eth32_open function.
- counter - Specifies the counter number (0 or 1).
- state - The new state for the specified counter. This may be:
  - COUNTER_DISABLED - The counter is disabled. The counter value may still be accessed, but the counter will not increment as a result of input signals.
  - COUNTER_FALLING - The counter will increment on the falling edge of the input signal.
  - COUNTER_RISING - The counter will increment on the rising edge of the input signal.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_get_counter_state, eth32_set_counter_rollover, eth32_set_counter_value

---

eth32_set_counter_threshold

```
int eth32_set_counter_threshold(eth32 handle, int counter, int threshold);
```

Summary

This function defines a counter event threshold that will cause an event to fire as the counter value passes the threshold. On the ETH32 device, only Counter 0 supports this (although both counters support rollover thresholds). An event is fired as a result of the counter surpassing the threshold, not meeting it. For example, with a threshold of 9, the counter's value would increment from 8 to 9 without firing the event, but it would fire as the counter incremented from 9 to 10. The valid range for a counter event threshold is from 0 to the maximum possible counter value (65535 for 16-bit counter 0). The powerup default threshold is 0. The threshold has no other side-effects on the counter - it does not reset the counter to 0 like the rollover threshold.

Parameters

- handle - The value returned by the eth32_open function.
- counter - Specifies the counter number. This must be 0.
- threshold - Specifies the event threshold for the counter (0-65535).

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

Remarks

Please note that defining a threshold with this function does not enable the current connection to actually receive the event notifications when they occur. These must be enabled separately using the eth32_enable_event function. Also note that event thresholds are common to all connections. Changing the thresholds will affect other connections if they are utilizing that particular counter event.

See Also

eth32_get_counter_threshold, eth32_set_counter_state, eth32_set_counter_value

---

eth32_set_counter_value

```
int eth32_set_counter_value(eth32 handle, int counter, int value);
```

Summary

This function loads a new value for the specified counter on the device. Since a counter is used to count the number of pulses / clocks from an external source, this function is typically not used frequently. It is useful for initializing the counter. All counters begin with a value of zero after powerup or reset.

Parameters

- handle - The value returned by the eth32_open function.
- counter - Specifies the counter number (0 or 1).
- value - Specifies the new value to load into the counter. For counter 0 (a 16-bit counter), this may be 0-65535. For counter 1 (an 8-bit counter), this may be 0-255.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_get_counter_value, eth32_set_counter_rollover, eth32_set_counter_state

---

eth32_set_direction

```
int eth32_set_direction(eth32 handle, int port, int direction);
```

Summary

This function sets the direction register for a digital I/O port, which configures each pin (bit) of the port as an input or output. The direction of each bit of the port can be set individually, meaning that some bits of the port can be inputs at the same time that other bits on the same port are outputs. A 1-bit in the direction register causes the corresponding bit of the port to be put into output mode, while a 0-bit specifies input mode. For example, a value of 0xF0 would put bits 0-3 into input mode and bits 4-7 into output mode.

Parameters

- handle - The value returned by the eth32_open function.
- port - The port number (0-5).
- direction - The new direction register for the port.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

Remarks

Port 3 shares its pins with the analog channels. When the ADC is enabled, all pins of port 3 are forced into input mode. The direction register of port 3 cannot be modified while the ADC is enabled.

The valid range for the direction parameter is any 8-bit number (ranges from 0 to 255). However, note that because ports 4 and 5 are single-bit ports, only bit 0 will have any effect on those ports.

For your convenience, constants for the direction parameter are provided that configure the port bits to be all inputs or all outputs. These are, respectively, DIR_INPUT and DIR_OUTPUT.

Example

```
eth32 handle;
int result;

// .... Your code that establishes a connection here

// Configure all odd bits of port 0 as inputs and even bits as outputs
//  Direction parameter of 10101010 binary, which is 0xAA hex or 170 decimal
result=eth32_set_direction(handle, 0, 0xAA);
if(result)
{
        // handle error
}
```

See Also

eth32_get_direction, eth32_get_direction_bit, eth32_set_direction_bit

---

eth32_set_direction_bit

```
int eth32_set_direction_bit(eth32 handle, int port, int bit, int direction);
```

Summary

This function alters a single bit of a port's direction register without affecting the value of any other bits. See the eth32_set_direction function for further description of the direction register.

Parameters

- handle - The value returned by the eth32_open function.
- port - The port number (0-5).
- bit - Which bit within the port to alter (0-7).
- direction - Make the bit an input (0) or an output (1).

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

Remarks

This function alters the specified direction register bit in a single operation directly on the ETH32 device. In other words, it does NOT read the current value over the network, modify it and then write it back. By doing it in a single operation, this avoids the potential of inadvertently overwriting changes made to other bits by other connections.

Port 3 shares its pins with the analog channels. When the ADC is enabled, all pins of port 3 are put into analog mode. The direction register of port 3 cannot be modified while the ADC is enabled.

See Also

eth32_set_direction, eth32_get_direction, eth32_get_direction_bit

---

eth32_set_eeprom

```
int eth32_set_eeprom(eth32 handle, int address, int length, void *buffer);
```

Summary

This function stores data into the non-volatile EEPROM memory of the device. Writing to EEPROM memory is a relatively slow process, which will temporarily disrupt event monitoring on the device. See the user manual for specific timing information.

Parameters

- handle - The value returned by the eth32_open function.
- address - The starting location into which data should be stored (0-255).
- length - The number of bytes to store. Valid values for this parameter depend on what is provided for the address parameter. For example, with an address of 0, you may specify a length of all 256 bytes, but with an address of 255, length may only be 1.
- buffer - The buffer containing the data to be stored. This must be at least as long as the number of bytes you have requested to store.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_get_eeprom

---

eth32_set_event_handler

```
int eth32_set_event_handler(eth32 handle, eth32_handler *handler);
```

Summary

This function configures an event handler mechanism within the API, which is able to immediately notify your code when events arrive. The event handler is separate from the event queue and enabling an event handler does not disable the event queue or vice versa.

Parameters

- handle - The value returned by the eth32_open function.
- handler - A pointer to an eth32_handler data structure which you have filled in with information about how events should be handled.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

Example

```
// This code shows an example of how to set up a
// callback function event handler.

// Somewhere in your code, define a callback function, which may be
// named anything you want.
// On Windows, it must be stdcall calling convention
void __stdcall event_fired(eth32 handle, eth32_event *event, void *extra)
{
        switch(event->id)
        {
                case 1000:
                        // React accordingly to Port 1, Bit 3 event
                        break;
                case 1001:
                        // React accordingly to Port 1, Bit 4 event
                        break;
        }
}



eth32 handle;
int result;
eth32_handler event_handler={0}; // Initialize all data in the structure to zero.

// .... Your code that establishes a connection here

// Set up our callback function as the event handler for this connection
event_handler.type=HANDLER_CALLBACK;
event_handler.maxqueue=1000;
event_handler.fullqueue=QUEUE_DISCARD_NEW;
event_handler.eventfn=event_fired; // Store the address of the callback
result=eth32_set_event_handler(handle, &event_handler);
if(result)
{
        // handle error
}

// Enable events on Port 1, bits 3 and 4
result=eth32_enable_event(handle, EVENT_DIGITAL, 1, 3, 1000);
if(result)
{
        // handle error
}

result=eth32_enable_event(handle, EVENT_DIGITAL, 1, 4, 1001);
```

```
if(result)
{
        // handle error
}
```

See Also

Event Callback Function, eth32_enable_event, eth32_get_event_handler

---

eth32_set_event_queue_config

```
int eth32_set_event_queue_config(eth32 handle, int maxsize, int fullqueue);
```

Summary

This function configures the maximum allowable size and the behavior of the event queue within the API. If a nonzero maximum size is configured for the event queue, the API will queue up all event notifications that arrive from the ETH32, allowing you to retrieve and remove them later using the eth32_dequeue_event function. If the number of events in the queue reaches the maximum size you have defined before your application has a chance to retrieve them, the API will stop adding events to the queue. At that point, either old events will be shifted out to make room for the new, or the new events will be ignored, depending on the behavior you have specified with the *fullqueue* parameter to this function. The event queue starts out disabled on each connection.

This queue is strictly within the API and calling this function does not modify anything on the ETH32 device itself.

Parameters

- handle - The value returned by the eth32_open function.
- maxsize - Specifies the maximum number of events that are allowed to be queued.
- fullqueue - When the queue is full and new events come in, this specifies whether the new events should be discarded (QUEUE_DISCARD_NEW) or the oldest event in the queue shifted out and discarded to make room for the new event at the end of the queue (QUEUE_DISCARD_OLD).

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

Example

```
eth32 handle;
int result;

// .... Your code that establishes a connection here
```

```
// Configure the event queue to hold up to 1,000 events.
// If the queue is ever full and more events arrive, discard
// the new events.
result=eth32_set_event_queue_config(handle, 1000, QUEUE_DISCARD_NEW);
if(result)
{
        // Handle error
}
```

See Also

eth32_dequeue_event, eth32_enable_event, eth32_get_event_queue_status

---

eth32_set_led

```
int eth32_set_led(eth32 handle, int led, int value);
```

Summary

This function allows you to control the state of the two LED's built into the ETH32 device.

Parameters

- handle - The value returned by the eth32_open function.
- led - Identifies which LED (0 or 1) to set.
- value - Any nonzero value turns on the LED and zero turns it off.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_get_led

---

eth32_set_pwm_base_period

```
int eth32_set_pwm_base_period(eth32 handle, int period);
```

Summary

This function configures the main PWM clock to have a cycle period of the given number of counts. This defines the base frequency that will be used for the PWM channels. The base frequency is not individually selectable for each channel, so this function will affect both PWM outputs. Each complete PWM waveform will have a duration of (period + 1) PWM clock cycles. The PWM clock counts at a rate of 2 MHZ. This means, for example, that specifying a period of 99 would result in a frequency of 20 KHZ (2,000,000/(99+1)). The base period is specified as a 16-bit number that may range from a value of 49 (40

KHZ) to a value of 65,535 (30.5 HZ).

Parameters

- handle - The value returned by the eth32_open function.
- period - Number of PWM clock counts to make up the base period of the PWM channels. This may range from 49 - 65535.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_get_pwm_base_period, eth32_set_pwm_channel, eth32_set_pwm_clock_state, eth32_set_pwm_duty_period,

---

eth32_set_pwm_channel

```
int eth32_set_pwm_channel(eth32 handle, int channel, int state);
```

Summary

This function configures the state of the specified PWM channel. When a channel is disabled, the I/O pin will function as a normal digital I/O pin. When the channel is enabled, that I/O pin will be overridden and the pin will become the PWM output. However, note that the pin must be put into output mode using the eth32_set_direction or eth32_set_direction_bit functions.

Parameters

- handle - The value returned by the eth32_open function.
- channel - Specifies the PWM channel number whose state should be set (0 or 1).
- state - Specifies the new state of the PWM channel. This may be:
  - PWM_CHANNEL_DISABLED - The PWM pin will function as a normal digital I/O pin.
  - PWM_CHANNEL_NORMAL - The PWM pin will function as a PWM output. It will be high for the time specified by the duty period and low for the rest of the PWM base period.
  - PWM_CHANNEL_INVERTED - The PWM pin will function as a PWM output. It will be low for the time specified by the duty period and high for the rest of the PWM base period.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_get_pwm_channel, eth32_set_pwm_base_period, eth32_set_pwm_clock_state, eth32_set_pwm_duty_period,

---

eth32_set_pwm_clock_state

```
int eth32_set_pwm_clock_state(eth32 handle, int state);
```

Summary

This function enables or disables the PWM clock from counting. The PWM clock is shared between both PWM outputs of the device. When the PWM clock is disabled, the PWM outputs will be idle (not pulsing). The PWM clock may be enabled or disabled independently of whether the individual PWM channel outputs are enabled or disabled.

Parameters

- handle - The value returned by the eth32_open function.
- state - This may be PWM_CLOCK_DISABLED to disable the clock or PWM_CLOCK_ENABLED to enable the clock.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_get_pwm_clock_state, eth32_set_pwm_base_period, eth32_set_pwm_channel, eth32_set_pwm_duty_period,

---

eth32_set_pwm_duty_period

```
int eth32_set_pwm_duty_period(eth32 handle, int channel, int period);
```

Summary

This function sets the duty period for a PWM channel, which is the length of time the PWM output is active during each PWM cycle. The duty period is specified as PWM clock counts less one. In other words, when the PWM channel state is in normal mode, the PWM output will be high for (period + 1) counts of the PWM clock and low for the remainder of the clock counts in the cycle. The length of the

PWM cycle is called the base period and set using the eth32_set_pwm_base_period function.

Parameters

- handle - The value returned by the eth32_open function.
- channel - Specifies the PWM channel number (0 or 1).
- period - Specifies the duty period for the channel, in terms of PWM clock counts (0-65535).

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

Remarks

Remember that the base period (set with eth32_set_pwm_base_period) is shared between both PWM channels on the device. However, the duty period (set with this function) is individually configurable for each channel. The recommended approach is to choose a PWM frequency that is appropriate for both channels and set the base period accordingly once during initialization. After that point, the individual duty periods for each channel should be set whenever necessary in order to alter the percentage of time the channel is on (duty cycle).

Any 16-bit value can be specified for the period, from 0 to 65535. Note that if a duty period is given that is greater than or equal to the current PWM base period, the output will be a solid high (in normal mode) or a solid low (in inverted mode). If a duty period of 0 is given, the output will not be solid, but rather it will have a short spike during each period of the PWM clock.

Example

```
// Error handling is omitted for clarity
eth32 handle;

// .... Your code that establishes a connection here


// Set up PWM channel 0 to have a 10 KHZ, 60% PWM signal:

// First, set up the base period to give a frequency of 10 KHZ
// This is calculated as:
//   (2,000,000)/(10,000) - 1
// We subtracted one since the base period takes one clock
// cycle longer than the value we load in.
eth32_set_pwm_base_period(handle, 199);

// Set up this PWM channel's duty period to take up 60% of
// each base period cycle.  The base period takes 200 clock
// cycles, so we want the duty period to take:
//   200 * 0.60 = 120 clock cycles
// Since the duty period takes one cycle longer than the value
// we load into it, we specify 119 here:
eth32_set_pwm_duty_period(handle, 0, 119);
```

```
// Put the PWM pin into output mode
// PWM 0's output pin is on Port 2, bit 4
eth32_set_direction_bit(handle, 2, 4, 1);

// Enable the main PWM clock
eth32_set_pwm_clock_state(handle, PWM_CLOCK_ENABLED);

// Finally, enable the PWM channel
eth32_set_pwm_channel(handle, 0, PWM_CHANNEL_NORMAL);
```

See Also

eth32_get_pwm_duty_period, eth32_set_pwm_base_period, eth32_set_pwm_channel, eth32_set_pwm_clock_state

---

eth32_set_pwm_parameters

```
int eth32_set_pwm_parameters(eth32 handle, int channel, int state, float freq, float duty);
```

Summary

This function is provided for your convenience in working with all of the various PWM settings. It allows you to specify the PWM frequency and the duty cycle of a channel in more familiar terms (hertz and percentage) rather than PWM clock counts. It also puts the appropriate I/O pin into output mode unless you specify that the PWM channel should be disabled. This function internally calls several other API functions to set everything up, therefore replacing calls to eth32_set_pwm_base_period, eth32_set_pwm_duty_period, eth32_set_pwm_clock_state, eth32_set_pwm_channel, and eth32_set_direction_bit with a single call to this function.

Parameters

- handle - The value returned by the eth32_open function.
- channel - Specifies the PWM channel number (0 or 1).
- state - Specifies the new state for the PWM channel. This may be PWM_CHANNEL_DISABLED, PWM_CHANNEL_NORMAL, or PWM_CHANNEL_INVERTED.
- freq - Specifies the frequency in Hertz. The valid range is 30.5 HZ to 40,000 HZ (40 KHZ)
- duty - Specifies the duty cycle as a percentage (A floating point number from 0.0 to 1.0). This specifies the percentage of each cycle that the channel will be active.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

Remarks

Note that this function calls eth32_set_pwm_base_period to set the PWM base period. Because the PWM base period is shared between both PWM channels, this will affect the other PWM channel if you specify a frequency different than what is already in effect.

Example

```
eth32 handle;
int result;

// .... Your code that establishes a connection here

// Set up PWM channel 0 to have a 10 KHZ, 60% PWM signal:
result=eth32_set_pwm_parameters(handle, 0, PWM_CHANNEL_NORMAL, 10000, 0.60);
if(result)
{
        // Handle error
}
```

See Also

eth32_get_pwm_parameters

---

eth32_set_timeout

```
int eth32_set_timeout(eth32 handle, unsigned int timeout);
```

Summary

This function is used to set the internal API timeout on any functions that require a response from the ETH32 device (for example, eth32_input_byte). If a function does not receive a reply within the timeout period specified, it returns the ETH_TIMEOUT error. This function does not affect the actual ETH32 device, but just the functionality within the API itself. This function does not affect any other open handles to devices.

Parameters

- handle - The handle value returned by the eth32_open function.
- timeout - Specifies the timeout in milliseconds. A value of zero means that functions should never time out.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_get_timeout

---

eth32_verify_connection

```
int eth32_verify_connection(eth32 handle);
```

Summary

This function sends a "ping" packet to the ETH32 device and waits for a response. It can be used to verify that there is still a good connection to the device.

Parameters

- handle - The value returned by the eth32_open function.

Return Value

This function returns zero on success (when a response is received within the timeout period) and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32_close, eth32_open, eth32_set_timeout

Event Callback Function

If you wish to use a callback function to handle events, you must write a callback event handler function in your code and instruct the API to call your function whenever an event arrives. You instruct the API to do this by setting up a callback event handler with the eth32_set_event_handler function.

The callback function is a function written by you, the programmer. Because it is a function you write, you have complete freedom to inspect whichever aspects of the event data you need to and react however you see fit.

Your callback function will be executed by a separate thread. You should be aware of this fact if you will be doing any tasks in your callback that are not thread safe. The API waits for your callback to return before calling it again with the next event. Therefore, be aware that if you perform any long operations within the callback, it will delay more events from being processed. Note that each connection handle has its own event thread, so if you are using a single callback function for multiple connections, be aware that at times there may be more than one instance of your callback function executing.

Callback Prototype and Parameters

Your callback function may be given any name, but regardless of its name, it must have one of the following prototypes, depending on your platform.

On Windows, you must use the standard calling convention (stdcall) as follows:

```
void __stdcall your_function_name(eth32 handle, eth32_event *event, void *extra);
```

On Linux, no calling convention modifier is needed:

```
void your_function_name(eth32 handle, eth32_event *event, void *extra);
```

If your application opens a connection to more than one device, you may still use the same callback function for all connections if you desire. Two parameters are passed to your callback that can be used to differentiate between connections: The handle for the connection on which the event occurred is passed to your function, and the *extra* member of the eth32_handler structure (as it was passed to the eth32_set_event_handler function) is also passed to your callback.

Of course, your callback is also provided with the details about the event that caused it to fire. The *event* parameter points to an eth32_event structure containing all of the event information. You should not modify any of the information contained in the structure.

Example

```
// This code shows an example of how to set up a
// callback function event handler.

// Somewhere in your code, define a callback function, which may be
// named anything you want.
// On Windows, it must be stdcall calling convention
void __stdcall event_fired(eth32 handle, eth32_event *event, void *extra)
{
        switch(event->id)
        {
                case 1000:
                        // React accordingly to Port 1, Bit 3 event
                        break;
                case 1001:
                        // React accordingly to Port 1, Bit 4 event
                        break;
        }
}



eth32 handle;
int result;
eth32_handler event_handler={0}; // Initialize all data in the structure to zero.

// .... Your code that establishes a connection here

// Set up our callback function as the event handler for this connection
event_handler.type=HANDLER_CALLBACK;
```

```
event_handler.maxqueue=1000;
event_handler.fullqueue=QUEUE_DISCARD_NEW;
event_handler.eventfn=event_fired; // Store the address of the callback
result=eth32_set_event_handler(handle, &event_handler);
if(result)
{
        // handle error
}

// Enable events on Port 1, bits 3 and 4
result=eth32_enable_event(handle, EVENT_DIGITAL, 1, 3, 1000);
if(result)
{
        // handle error
}

result=eth32_enable_event(handle, EVENT_DIGITAL, 1, 4, 1001);
if(result)
{
        // handle error
}
```

Configuration / Detection

The ETH32 API includes a number of functions for use in detecting and configuring ETH32 devices. These functions and associated structures are described below.

Error Handling

As with the main API, most configuration and detection functions return error codes. Zero indicates no error, while a negative error code indicates that an error occurred.

Structures

eth32cfg_ip_t Structure

The eth32cfg_ip_t structure holds an IP address in binary form. It is used to represent IP address information in the ETH32 device configuration structure, to specify the broadcast address, and to retrieve IP address information about the PC's network interfaces.

```
typedef struct
{
        unsigned char byte[4];
} eth32cfg_ip_t;
```

- byte - Array containing individual octets of the IP address. Index 0 contains the most significant, e.g. 192 from the address 192.168.1.100

eth32cfg_data_t Structure

The eth32cfg_data_t structure holds all of the network configuration and device information data for a particular ETH32 device. It is provided to your application when retrieving information about detected devices. Your application can also fill in or modify the information and provide it to the API to store new configuration into a device.

```
typedef struct
{
        unsigned char product_id;
        unsigned char firmware_major;
        unsigned char firmware_minor;
        unsigned char config_enable;
        unsigned char mac[8];
        unsigned short serialnum_batch;
        unsigned short serialnum_unit;
        eth32cfg_ip_t config_ip;
        eth32cfg_ip_t config_gateway;
        eth32cfg_ip_t config_netmask;
        eth32cfg_ip_t active_ip;
        eth32cfg_ip_t active_gateway;
        eth32cfg_ip_t active_netmask;
        unsigned char dhcp;

} eth32cfg_data_t;
```

- product_id - Contains the product ID code for the device. This will be 105 for ETH32 devices. This makes up a portion of the serial number.
- firmware_major - Contains the major portion of the firmware version, e.g. 3 from 3.000
- firmware_minor - Contains the minor portion of the firmware version, e.g. 0 from 3.000
- config_enable - Nonzero if the device’s Allow Config switch is set to Yes
- mac - The MAC address of the device. Not that this is an eight-element array. Only the first six are used, and the last two are for proper structure alignment.
- serialnum_batch - The batch number portion of the device’s serial number
- serialnum_unit - The unit number portion of the device’s serial number
- config_ip - The static IP address stored in the device. This is ignored if DHCP is active.
- config_gateway - The static gateway IP address stored in the device. This is ignored if DHCP is active.
- config_netmask - The static network mask stored in the device. This is ignored if DHCP is active.
- active_ip - The IP address being used by the device, whether it was provided by DHCP or statically configured.

- active_gateway - The gateway IP address being used by the device, whether it was provided by DHCP or statically configured.
- active_netmask - The network mask being used by the device, whether it was provided by DHCP or statically configured.
- dhcp - Nonzero if DHCP is being used by the device, or zero if the static settings (config_ip, etc) are being used.

If a device is using DHCP, then active_ip will most likely be different than the static (stored) config_ip, and so on for the gateway and netmask. If DHCP is not being used, then active_ip will be the same as config_ip, and so on for the gateway and netmask.

When using this structure with the eth32cfg_set_config, you may modify the config_ip, config_gateway, config_netmask, and dhcp members in order to update the corresponding settings within the ETH32 device. The other members of the structure should not be modified, since they will either be ignored, or are required for the new configuration to be accepted by the device. Specifically, the MAC address and serial number information must match the device's information, or the device will ignore the new configuration data.

Configuration / Detection Function Reference

eth32cfg_discover_ip

```
eth32cfg eth32cfg_discover_ip(eth32cfg_ip_t *bcastaddr, unsigned int flags, unsigned char *mac,
              unsigned char product_id, unsigned short serialnum_batch,
              unsigned short serialnum_unit, int *number, int *result);
```

Summary

This function is used to detect ETH32 devices and their currently-active IP configuration settings. This function allows you to specify filter flags so that only the information for the specific ETH32 device that you are interested in will be returned (in case there are multiple ETH32s on the network). This is intended for applications that need to discover the IP of a device that is using DHCP to get its IP address. This function uses a new command to the ETH32 device that is only supported by devices with firmware v3.000 and on. Any older devices on the network will not be detected. The eth32cfg_data_t structure for devices detected with this function will not have all fields filled in, since the response from the ETH32 does not include all available information. Only the product_id, mac, serialnum_batch, serialnum_unit, active_ip, active_gateway, active_netmask, and dhcp fields will be filled in and valid.

The flags parameter instructs the function which data to filter on. Although this function includes parameters for both MAC and serial number information, they will only be considered if the appropriate flag is present in the flags parameter.

Once this function returns, the configuration data for any devices that have been found will be available through the eth32cfg_get_config function. When you are finished with the results, they should be freed using eth32cfg_free.

Parameters

- bcastaddr - Broadcast address to which queries should be sent. Passing in NULL will use 255.255.255.255, which is suitable for most situations.
- flags - Specifies which parameters should be considered in discovering the device. If more than one flag is specified, then the device must match BOTH. This parameter may be one or a combination of the following values:
  - ETH32CFG_FILTER_NONE - The parameters will be ignored. All devices will be discovered.
  - ETH32CFG_FILTER_MAC - Only devices matching the provided MAC address will be discovered.
  - ETH32CFG_FILTER_SERIAL - Only devices matching the provided serial number information (id, batch, unit) will be discovered.
- mac - The MAC address of the device you are trying to discover. If flags does not include ETH32CFG_FILTER_SERIAL, this can be NULL.
- product_id - The product ID code (part of the serial number) of the device you are trying to discover. For ETH32 devices, this is 105.
- serialnum_batch - The batch number portion of the serial number for the device you are trying to discover.
- serialnum_unit - The unit number portion of the serial number for the device you are trying to discover.
- number - A pointer to an integer which will receive the number of devices that were found.
- result - A pointer to an integer which will receive an error code. If the function returns a nonzero handle, this value will be zero.

Return Value

The return type is defined as eth32cfg, which is a handle typedef'ed as a void pointer. This function returns a nonzero handle on success, or zero on failure. In case of failure, the specific error code is stored into the result parameter, if provided. A valid returned handle can be used with the eth32cfg_get_config function to retrieve device information.

Remarks

If no devices are found, but no error has occurred, the function will still return a nonzero handle, but also indicate in the number parameter that 0 devices were found. Even in this case, the handle must be freed using eth32cfg_free.

Example

```
eth32 dev;
eth32cfg handle;
eth32cfg_ip_t bcast;
eth32cfg_data_t devdata;
char buf[50];
int number;
int result;

// We could just pass a null pointer in for the broadcast address, but
// we'll show how to define a broadcast address here.
eth32cfg_string_to_ip("255.255.255.255", &bcast);
// We could also do bcast.byte[0]=255; and so on through byte[3]

// Find a device by serial number -- we can use the ETH32_PRODUCT_ID constant,
// 1 for the batch (AB), and 82 for the unit number.
// This would be serial number 105-AB082 as shown on the device.
handle=eth32cfg_discover_ip(&bcast, ETH32CFG_FILTER_SERIAL, NULL,
                            ETH32_PRODUCT_ID, 1, 82,
                            &number, &result);

if(result)
{
        printf("Error detecting device: %s\n", eth32_error_string(result));
        // handle error as appropriate in your code, prevent falling through
        // to code below.
}

if(number==0)
{
        printf("Device not found.\n");
}
else
{
        // Retrieve all the device information into our structure
        eth32cfg_get_config(handle, 0, &devdata);
        // Convert the Active IP into a string
        eth32cfg_ip_to_string(&(devdata.active_ip), buf);

        // Now connect to the device and turn on LED 0
        // Error checking omitted for brevity
        dev=eth32_open(buf, ETH32_PORT, 0, &result);
        eth32_set_led(dev, 0, 1);
        eth32_close(dev);

}

// Free the results when finished
eth32cfg_free(handle);
```

See Also

eth32cfg_get_config, eth32cfg_free, eth32cfg_discover_ip

eth32cfg_free

```
void eth32cfg_free(eth32cfg handle);
```

Summary

This function frees any memory associated with the current set of results held by the provided handle. This must be called after you are finished with the results from eth32cfg_discover_ip or eth32cfg_query.

Parameters

- handle - The value returned by eth32cfg_discover_ip or eth32cfg_query.

Return Value

This function does not return a value.

See Also

eth32cfg_discover_ip, eth32cfg_query

eth32cfg_get_config

```
int eth32cfg_get_config(eth32cfg handle, int index, eth32cfg_data_t *dataptr);
```

Summary

This function is used to access the device information and configuration data for each device that was found using eth32cfg_discover_ip or eth32cfg_query.

Parameters

- handle - The value returned by eth32cfg_discover_ip or eth32cfg_query.
- index - The index of the result to return.
- dataptr - Pointer to the structure to be filled in the the device information.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

Remarks

The index is zero-based, which means it can range from zero up to one less than the number of available results.

See Also

eth32cfg_data_t Structure, eth32cfg_discover_ip, eth32cfg_query

eth32cfg_ip_to_string

```
int eth32cfg_ip_to_string(const eth32cfg_ip_t *ipbinary, char *ipstring);
```

Summary

This function converts the eth32cfg_ip_t binary representation into a string. You must provide an valid buffer of at least 16 bytes to the function where the string will be written. The function will null-terminate the string.

Parameters

- ipbinary - The IP address to be converted.
- ipstring - Pointer to a string buffer where the string representation will be written. This buffer must be at least 16 bytes long or memory corruption could occur.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32cfg_string_to_ip

eth32cfg_query

```
eth32cfg eth32cfg_query(eth32cfg_ip_t *bcastaddr, int *number, int *result);
```

Summary

This function is used to detect all ETH32 devices on the local network segment and all of their available device information and configuration settings. Once this function returns, the configuration data for any devices that have been found will be available through the eth32cfg_get_config function. When you are finished with the results, they must be freed using eth32cfg_free.

Parameters

- bcastaddr - Broadcast address to which queries should be sent. Passing in NULL will use 255.255.255.255, which is suitable for most situations.
- number - A pointer to an integer which will receive the number of devices that were found.
- result - A pointer to an integer which will receive an error code. If the function returns a nonzero handle, this value will be zero.

Return Value

The return type is defined as eth32cfg, which is a handle typedef'ed as a void pointer. This function returns a nonzero handle on success, or zero on failure. In case of failure, the specific error code is stored into the result parameter, if provided. A valid returned handle can be used with the eth32cfg_get_config function to retrieve device information.

Remarks

If no devices are found, but no error has occurred, the function will still return a nonzero handle, but also indicate in the number parameter that 0 devices were found. Even in this case, the handle must be freed using eth32cfg_free.

As opposed to the eth32cfg_discover_ip function, which is only supported by devices with firmware 3.000 and greater, the eth32cfg_query function detects all devices with all firmware versions. This function sends several queries out repeatedly in case any queries or responses are lost on the network. It also delays for a short while to listen for responses. Because of this, the eth32cfg_discover_ip function will be faster if you are looking for a specific device, know its MAC address or serial number, and know it is running firmware v3.000 or greater.

Example

```
eth32cfg handle;
eth32cfg_ip_t bcast;
eth32cfg_data_t devdata;
char buf[50];
int number;
int result;
int i;

// We could just pass a null pointer in for the broadcast address, but
// we'll show ways to define a broadcast address here.
eth32cfg_string_to_ip("255.255.255.255", &bcast);
// We could also do bcast.byte[0]=255; and so on through byte[3]

handle=eth32cfg_query(&bcast, &number, &result);

if(result)
{
        printf("Error detecting devices: %s\n", eth32_error_string(result));
        // handle error as appropriate in your code, prevent falling through
        // to code below.
```

```
}

if(number==0)
{
        printf("No devices were found.\n");
}
else
{
        for(i=0; i<number; i++)
        {
                // Retrieve all the device information into our structure
                eth32cfg_get_config(handle, i, &devdata);
                // Convert the Active IP into a string
                eth32cfg_ip_to_string(&(devdata.active_ip), buf);
                // And print it out
                printf("Device found with IP address of: %s\n", buf);
        }
}

// Free the results when finished
eth32cfg_free(handle);
```

See Also

eth32cfg_get_config, eth32cfg_free, eth32cfg_discover_ip

eth32cfg_serialnum_string

```
int eth32cfg_serialnum_string(unsigned char product_id, unsigned short batch, unsigned short unit,
                              char *serialstring, int bufsize);
```

Summary

This function takes the numeric components of the ETH32 serial number and formats a serial number string in the same way that it is printed on the ETH32 device enclosure.

Parameters

- product_id - The product ID portion of the serial number
- batch - The batch number portion of the serial number
- unit - The unit number portion of the serial number
- serialstring - Pointer to a string buffer where the string representation will be written.
- bufsize - Specifies how many bytes long the serialstring buffer is.

Return Value

This function returns zero on success and a negative error code on failure. If the buffer size according to the bufsize parameter is not long enough to receive the entire serial number string, it will not be written, and the function will return the ETH_BUFSIZE error.

See Also

eth32cfg_data_t Structure

eth32cfg_set_config

```
int eth32cfg_set_config(eth32cfg_ip_t *bcastaddr, eth32cfg_data_t *dataptr);
```

Summary

This function is used to store new configuration settings into an ETH32 device. The device's Allow Config switch must be set to Yes, or the new configuration will be rejected.

Parameters

- bcastaddr - Broadcast address to which the configuration packet should be sent. Passing in NULL will use 255.255.255.255, which is suitable for most situations.
- dataptr - Pointer to the structure containing the new configuration data and product identification information.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes. Under normal circumstances, the device will accept the configuration and return a confirmation packet, which will cause the function to immediately return without an error. If the device's Allow Config switch is set to No, it will return a rejection packet, which will cause the function to return the ETH_CFG_REJECT error. If no response is received from the device, the function will return the ETH_CFG_NOACK error after a short timeout.

Remarks

The MAC address and serial number information members of the eth32cfg_data_t Structure identify which device is to be configured. If those members are not set correctly, the device will simply ignore the settings, or worst-case, if they match a different device you were not intending to configure, that device will accept the new configuration. Therefore, in most cases, although it is not required, it is best to take the eth32cfg_data_t Structure from the eth32cfg_get_config function, modify as needed, and then provide that to this function.

See Also

eth32cfg_data_t Structure, eth32cfg_get_config

eth32cfg_string_to_ip

```
int eth32cfg_string_to_ip(const char *ipstring, eth32cfg_ip_t *ipbinary);
```

Summary

This function converts a string representation of an IP address into the eth32cfg_ip_t binary representation of an IP address. If the string doesn't contain a valid IP address, an ETH_INVALID_IP error will be returned.

Parameters

- ipstring - The IP address to be converted.
- ipbinary - Pointer to an eth32cfg_ip_t structure which will be filled in with the converted IP address.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32cfg_ip_to_string

Plugin Function Reference

eth32cfg_plugin_choose_interface

```
int eth32cfg_plugin_choose_interface(eth32cfgiflist handle, int index);
```

Summary

This function selects one of the available network interfaces on the PC as the interface on which the ETH32 Configuration / Detection API should sniff for responses from ETH32 devices. This does not affect the main API functionality. The interface list must have been previously obtained using the eth32cfg_plugin_interface_list function and the provided index must be a valid index within that list. Currently, this function is only applicable when the WinPcap plugin is loaded. Otherwise, the ETH_NOT_SUPPORTED error will be returned.

Parameters

- handle - The value returned by eth32cfg_plugin_interface_list

- index - The index of the interface in the previously-obtained interface list which should be chosen for sniffing responses

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

See Also

eth32cfg_plugin_interface_list

eth32cfg_plugin_interface_address

```
int eth32cfg_plugin_interface_address(eth32cfgiflist handle, int index,
                                      eth32cfg_ip_t *ip, eth32cfg_ip_t *netmask);
```

Summary

This function retrieves IP address information about one of the interfaces in the network interface list that was previously obtained by calling eth32cfg_plugin_interface_list.

Parameters

- handle - The value returned by eth32cfg_plugin_interface_list
- index - The index of the interface within the list
- ip - Pointer to an eth32cfg_ip_t structure which will receive the IP address of the specified network interface
- netmask - Pointer to an eth32cfg_ip_t structure which will receive the network mask of the specified network interface

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

Remarks

The index is zero-based, which means it can range from zero up to one less than the number of available interfaces.

See Also

eth32cfg_plugin_interface_list, eth32cfg_plugin_interface_list_free

eth32cfg_plugin_interface_list

```
eth32cfgiflist eth32cfg_plugin_interface_list(int *numd, int *result);
```

Summary

This function loads the list of available network interface cards on the PC. A plugin which provides this functionality must be loaded first before calling this function. This functionality is provided by both the System and the WinPcap plugins, but not by the None plugin. Once the function returns, details of each interface can be accessed using eth32cfg_plugin_interface_address, eth32cfg_plugin_interface_type, and eth32cfg_plugin_interface_name. Once you are done with the network interface list, the memory used by the interface list must be freed with eth32cfg_plugin_interface_list_free. This must be done before changing the plugin with eth32cfg_plugin_load.

Parameters

- numd - Pointer to a variable which will receive the number of network interfaces in the resulting list.
- result - Pointer to a variable which will receive an error code.

Return Value

The return type is defined as eth32cfgiflist, which is a handle typedef'ed as a void pointer. This function returns a nonzero handle on success, or zero on failure. However, if the function returns zero and the result code is also zero (indicating success), then it simply means that no network interfaces were found. In case of failure, the specific error code is stored into the result parameter, if provided. A valid nonzero handle can be used with other functions to obtain details about the network interfaces.

Remarks

If the currently-loaded plugin does not provide this functionality, an ETH_NOT_SUPPORTED error will be stored into result, and zero will be returned.

See Also

eth32cfg_plugin_interface_list_free, eth32cfg_plugin_interface_address, eth32cfg_plugin_interface_type, eth32cfg_plugin_interface_name

eth32cfg_plugin_interface_list_free

```
void eth32cfg_plugin_interface_list_free(eth32cfgiflist handle);
```

Summary

This function frees the memory associated with the network interface list that was previously obtained by calling eth32cfg_plugin_interface_list. You must call this function on any open interface list handles in the same application process before loading a different plugin with eth32cfg_plugin_load.

Parameters

- handle - The value returned by eth32cfg_plugin_interface_list

Return Value

This function does not return a value.

See Also

eth32cfg_plugin_interface_list

eth32cfg_plugin_interface_name

```
int eth32cfg_plugin_interface_name(eth32cfgiflist handle, int index,
                                   int nametype, char *name, int *length);
```

Summary

This function retrieves name and description information about one of the interfaces in the network interface list that was previously obtained by calling eth32cfg_plugin_interface_list. Depending on the plugin that is currently loaded, each interface may have several types of names available. This function needs to be called separately for each type of name you want to retrieve.

Parameters

- handle - The value returned by eth32cfg_plugin_interface_list
- index - The index of the interface within the list
- nametype - This can be one of the following values:
    - ETH32CFG_IFACENAME_STANDARD - Retrieves a string which is typically an internal identifier string for the interface, but is not very human-readable. The exact value depends on the plugin being used. This string will be available when using the System plugin or when using the WinPcap plugin.
    - ETH32CFG_IFACENAME_FRIENDLY - Retrieves the human-readable name for the interface. For example, Local Area Connection. The ETH_NOT_SUPPORTED error will be returned if the WinPcap plugin is loaded.
    - ETH32CFG_IFACENAME_DESCRIPTION - Retrieves a description of the interface. The value of the string depends on the plugin being used, but typically includes the manufacturer or model of the card. The string will be available when using the System plugin or when using the WinPcap plugin.
- name - Pointer to string buffer where the requested name/description should be stored

- length - Pointer to variable containing the length of the name buffer. You must store the buffer length to this variable before calling this function. If the buffer is not large enough and the function returns ETH_BUFSIZE, then the function will also store the required buffer length into this variable.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes.

Remarks

The index is zero-based, which means it can range from zero up to one less than the number of available interfaces.

See Also

eth32cfg_plugin_interface_list, eth32cfg_plugin_interface_list_free

eth32cfg_plugin_interface_type

```
int eth32cfg_plugin_interface_type(eth32cfgiflist handle, int index, int *type);
```

Summary

This function retrieves the interface type for one of the interfaces in the network interface list that was previously obtained by calling eth32cfg_plugin_interface_list.

Parameters

- handle - The value returned by eth32cfg_plugin_interface_list
- index - The index of the interface within the list
- type - Pointer to variable which will receive the interface type. This will be one of the following values:
    - ETH32CFG_IFTYPE_OTHER - This is used if the plugin provides information about the interface type, but it isn't one of the predefined constants.
    - ETH32CFG_IFTYPE_ETHERNET - Ethernet interface
    - ETH32CFG_IFTYPE_TOKENRING - Token Ring interface
    - ETH32CFG_IFTYPE_FDDI - FDDI (Fiber Distributed Data Interface) interface
    - ETH32CFG_IFTYPE_PPP - PPP (Point-to-Point Protocol) interface
    - ETH32CFG_IFTYPE_LOOPBACK - Local loopback interface (e.g. 127.0.0.1)

- ETH32CFG_IFTYPE_SLIP - SLIP (Serial Line Internet Protocol) interface

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes. If the currently loaded plugin does not provide interface type information, this function will return ETH_NOT_SUPPORTED.

Remarks

The index is zero-based, which means it can range from zero up to one less than the number of available interfaces.

See Also

eth32cfg_plugin_interface_list, eth32cfg_plugin_interface_list_free

eth32cfg_plugin_load

```
int eth32cfg_plugin_load(int option);
```

Summary

This function loads one of the pre-defined plugins. The currently-loaded plugin affects the entire process in terms of the Configuration and Detection functionality, but does not affect the main functionality of the API. See the Plugins topic for more information.

Parameters

- option - The plugin to be loaded. This can be one of the following options:
  - ETH32CFG_PLUG_NONE - No plugin loaded. This is the default if Load is never called. If another plugin is loaded, calling Load with this option will remove the loaded plugin.
  - ETH32CFG_PLUG_SYS - System plugin. The Windows API is used to provide information about the network interfaces on the PC. Using this plugin does not affect how queries are sent out or how responses are received.
  - ETH32CFG_PLUG_PCAP - WinPcap plugin. The WinPcap library is used to provide information about the network interfaces as well as to sniff for ETH32 responses on the chosen interface.

Return Value

This function returns zero on success and a negative error code on failure. Please see the Error Codes section for possible error codes. If a plugin is attempted to be loaded that is not present on the system, an ETH_NOT_SUPPORTED error will be returned.

When one plugin (other than NONE) has been loaded and any interface list handles returned by eth32cfg_plugin_interface_list are open, you must make sure that the eth32cfg_plugin_interface_list_free function is called for each of them before changing the plugin with this function. This is due to the fact that the loaded plugin affects the entire process, so it is up to you as the programmer to ensure that any open interface lists are freed before changing the plugin.

See Also

eth32cfg_plugin_interface_list_free

.NET Languages

The .NET class provided by the ETH32 distribution is a "wrapper" class in that it depends on the core ETH32 API (eth32api.dll) for almost every action. While it internally uses the core API, it provides a more convenient way to use the API within .NET languages. Particularly in the area of event processing, this class takes care of some of the behind-the-scenes details to make event handling easy to implement and consistent with the facilities provided by the language.

Unless otherwise specified, the code declarations and examples provided throughout this section are in C# syntax. Other .NET languages will require slightly different syntax, but have exactly the same parameters, return values, and behavior.

Getting Started

In .NET languages, you must add a reference to the Eth32.dll assembly. In the Visual Studio C# and VB.NET environments, this can be done by following these steps:

- On the Project menu, select "Add Reference".
- Click the Browse button. Find the Eth32.dll file from the ETH32 API distribution. The default installation directory for this file is C:\Program Files\winford\eth32\api\windows\ms.net. This file is also on the CD in the api\windows\ms.net directory.

Note that adding the reference causes Visual Studio to automatically copy the Eth32.dll file into your project directory.

In Managed C++, you must first manually copy the Eth32.dll file from the ETH32 API distribution (default location is C:\Program Files\winford\eth32\api\windows\ms.net) into your project directory (typically the same directory your source code files for that project are in). Then, you must include this line near the top of your managed C++ source file:

```
#using <Eth32.dll>
```

At this point, you have provided the compiler with enough information to successfully compile your application. In order to allow your application to run after compiling it, you also must place a copy of the Eth32.dll file into the subdirectory of your project where your executable is built, typically the Debug or Release directory. Those directories may not yet exist if you have not yet compiled your project.

Namespace

Although not required, you will most likely want to add a line to your source file to use the WinfordEthIO namespace, which contains all of the classes and definitions in the Eth32.dll assembly. This line should go near the top of your source file, and varies depending on language:

- Visual C#.NET

```
using WinfordEthIO;
```

- Visual Basic.NET

```
Imports WinfordEthIO
```

- Visual C++.NET (Managed)

```
using namespace WinfordEthIO;
```

If you decide not to include a namespace line, you will need to prepend WinfordEthIO before class names or other definitions you use from the assembly. Example code provided in this documentation assumes that the namespace line is used.

Basic Declaration

The main class provided by the assembly is the Eth32 class. Once an instance of this class has been created, you can begin using the members of this class, starting with Connect to create a connection. The basic code is typically as follows (substituting any valid variable name for dev and the actual address or DNS name of your device for 192.168.1.100):

- Visual C#.NET

```
Eth32 dev = new Eth32();
dev.Connect("192.168.1.100");
```

- Visual Basic.NET

```
Dim dev As New Eth32()
dev.Connect("192.168.1.100")
```

- Visual C++.NET (Managed)

```
Eth32 *eth = new Eth32;
eth->Connect("192.168.1.100");
```

Error Handling

Errors that may occur within the class or core API are provided to your application as exceptions. This means that as you use the device, you do not need to check return values for error codes. Instead, if an error occurs, an exception will be raised and the applicable error handling code you have designated (if any) will be executed. As a rule, you should include error handling code for your application so that, for example, if an attempt to connect to the device fails, it does not cause an unhandled exception (which

causes the application to exit).

The exception class used in .NET languages is Eth32Exception. This class inherits from the System.Exception class and adds the ErrorCode member. The ErrorCode member contains one of the possible values of the EthError enumerator, indicating more specifically what caused the error. The following example in VB.NET illustrates the basic idea.

```
Dim dev As New Eth32()

Try
        dev.Connect("192.168.1.100")
Catch my_error As Eth32Exception
        MsgBox("Error occurred involving the ETH32 device: " & dev.ErrorString(my_error.ErrorCode))
End Try
```

Error Codes

Error code constants are defined by the EthError enumerator. The following error codes are defined:

- EthError.None - Success, no error.
- EthError.General - A miscellaneous or uncategorized error has occurred.
- EthError.Closing - Function aborted because the device is being closed.
- EthError.Network - Network communications error. Connection was unable to be established or existing connection was broken.
- EthError.Thread - Internal error occurred in the threads and synchronization library.
- EthError.NotSupported - Function not supported by this device.
- EthError.Pipe - Internal API error dealing with data pipes.
- EthError.Rthread - Internal API error dealing with the "Reader thread."
- EthError.Ethread - Internal API error dealing with the "Event thread."
- EthError.Malloc - Error dynamically allocating memory.
- EthError.Windows - Internal API error specific to the Microsoft Windows platform.
- EthError.Winsock - Internal API error in dealing with the Microsoft Winsock library.
- EthError.NetworkIntr - Network read/write operation was interrupted.
- EthError.WrongMode - Something is not configured correctly in order to allow this functionality.
- EthError.BcastOpt - Error setting the SO_BROADCAST option on a socket.

- EthError.ReuseOpt - Error setting the SO_REUSEADDR option on a socket.
- EthError.ConfigNoack - Warning - device did not acknowledge our attempt to store a new configuration.
- EthError.ConfigReject - Device has rejected the new configuration data we attempted to store. Configuration switch on device may be disabled.
- EthError.Loadlib - Error loading an external DLL library.
- EthError.Plugin - General error with the currently configured plugin/sniffer library.
- EthError.Bufsize - A buffer provided was either invalid size or too small.
- EthError.InvalidHandle - Invalid device handle was given.
- EthError.InvalidPort - The given port number does not exist on this device.
- EthError.InvalidBit - The given bit number does not exist on this port.
- EthError.InvalidChannel - The given channel number does not exist on this device.
- EthError.InvalidPointer - The pointer passed in to an API function was invalid.
- EthError.InvalidOther - One of the parameters passed in to an API function was invalid.
- EthError.InvalidValue - The given value is out of range for this I/O port, counter, etc.
- EthError.InvalidIp - The IP address provided was invalid.
- EthError.InvalidNetmask - The subnet mask provided was invalid.
- EthError.InvalidIndex - Invalid index value.
- EthError.Timeout - Operation timed out before it could be completed.
- EthError.AlreadyConnected - An object that is already connected cannot have Connect called again.
- EthError.NotConnected - This operation requires the object to be connected.

Structures

eth32_event

The eth32_event structure holds all of the information about an event that has fired. It is included in the arguments to your event handler function (see the Event Handler section).

```
public struct eth32_event
{
        int id;
        int type;
        int port;
        int bit;
        int prev_value;
        int value;
        int direction;
} eth32_event;
```

- id - The user-assigned event ID that you gave this event when enabling it.
- type - Event type, as defined by the Eth32EventType enumerator constants Digital, Analog, CounterRollover CounterThreshold, and Heartbeat.
- port - For digital events, this specifies the port number the event occurred on. For analog events, it specifies the event bank number (0 or 1), and for counter events, it specifies which counter the event occurred on.
- bit - For a digital bit event, this specifies the bit number that changed. For an analog event, it specifies the analog channel, and for a digital port event, this will be -1.
- prev_value - The old value of the bit, port, or analog channel (as appropriate) before the event fired.
- value - The new value of the bit, port, or analog channel that caused the event to fire. In the case of counter events, this indicates the number of times the event occurred since the last time this event was fired (almost always 1).
- direction - Indicates whether the new value of the bit, port, or channel is greater or less than the previous value. It is 1 for greater than or -1 for less than.

Eth32 Member Reference

The members of the Eth32 class are described below. Several features of the ETH32 device are represented in the class as properties. When the property is read by your code, a request is sent to the ETH32 device, the ETH32 replies with the requested value, and that value is returned as the value of the property. When your code writes a new value to the property, a command is sent to the ETH32 storing the new value for that setting. All of the Eth32 class properties are both readable and writable unless otherwise specified.

AnalogAssignment Property

```
Eth32AnalogChannel AnalogAssignment[int channel]
```

Summary

When this property is written, it assigns a logical analog channel to one of the physical channels. When it is read, it returns the current physical channel assignment for the specified logical channel. The logical channel assignment specifies which physical pins are used to determine the value of the analog reading when that logical channel is read or monitored for events. There are eight logical channels, each of which may be arbitrarily assigned to physical channels using this property.

Parameters

- channel - The logical channel number (0-7).

Value

This property is a Eth32AnalogChannel value, which is an enumerator containing constants defining the possible channel assignments. The possible values of this enumerator are defined in the Remarks section below.

Remarks

The logical channels simply provide a way to select which of the many physical channel sources listed below will be continually updated for reading on the device and, if configured to do so, monitored for analog events.

The assignments given to the logical channels may be completely arbitrary. Also, it is permissible to have more than one logical analog channel assigned to the same physical channel source. This can occasionally be advantageous for event monitoring. Since there are two possible event definitions per logical channel, assigning more than one logical channel to the same physical channel allows more than two event definitions on that physical channel.

When the device is first powered up or the ResetDevice Method is called, the logical channel assignments revert to their defaults. Logical channel 0 is assigned to single-ended channel 0, logical channel 1 to single-ended channel 1 and so on.

The assignments made with this property are effective until they are either overwritten by setting the property again or the board is reset (hard reset or by calling the ResetDevice Method). There is no limitation on how often you may reassign logical channels.

The following constants are defined in the Eth32AnalogChannel enumerator. These are the valid physical channel sources to which a logical channel may be assigned. The constant definition should typically be used in your source code, but its hexadecimal value is shown for reference.

For single-ended channels, the reading comes from the voltage of the specified pin with respect to ground.

Table 4. Single-Ended Channels

Constant	Value	Physical Pin
Eth32AnalogChannel.SE0	0x00	Port 3, Bit 0
Eth32AnalogChannel.SE1	0x01	Port 3, Bit 1
Eth32AnalogChannel.SE2	0x02	Port 3, Bit 2
Eth32AnalogChannel.SE3	0x03	Port 3, Bit 3
Eth32AnalogChannel.SE4	0x04	Port 3, Bit 4
Eth32AnalogChannel.SE5	0x05	Port 3, Bit 5
Eth32AnalogChannel.SE6	0x06	Port 3, Bit 6
Eth32AnalogChannel.SE7	0x07	Port 3, Bit 7

For differential channels, the reading comes from the voltage difference between two pins. It is permissible for either to be positive or negative with respect to the other. They are simply labeled positive and negative inputs to specify how the reading is determined. Please note that the voltage on each pin must still remain within the range of 0 to 5V with respect to the ground of the device.

Table 5. Differential Channels

Constant	Value	Positive Input	Negative Input	Gain
Eth32AnalogChannel.DI00X10	0x08	Port 3, Bit 0	Port 3, Bit 0	10x
Eth32AnalogChannel.DI10X10	0x09	Port 3, Bit 1	Port 3, Bit 0	10x
Eth32AnalogChannel.DI00X200	0x0A	Port 3, Bit 0	Port 3, Bit 0	200x
Eth32AnalogChannel.DI10X200	0x0B	Port 3, Bit 1	Port 3, Bit 0	200x
Eth32AnalogChannel.DI22X10	0x0C	Port 3, Bit 2	Port 3, Bit 2	10x
Eth32AnalogChannel.DI32X10	0x0D	Port 3, Bit 3	Port 3, Bit 2	10x
Eth32AnalogChannel.DI22X200	0x0E	Port 3, Bit 2	Port 3, Bit 2	200x
Eth32AnalogChannel.DI32X200	0x0F	Port 3, Bit 3	Port 3, Bit 2	200x
Eth32AnalogChannel.DI01X1	0x10	Port 3, Bit 0	Port 3, Bit 1	1x
Eth32AnalogChannel.DI11X1	0x11	Port 3, Bit 1	Port 3, Bit 1	1x
Eth32AnalogChannel.DI21X1	0x12	Port 3, Bit 2	Port 3, Bit 1	1x
Eth32AnalogChannel.DI31X1	0x13	Port 3, Bit 3	Port 3, Bit 1	1x
Eth32AnalogChannel.DI41X1	0x14	Port 3, Bit 4	Port 3, Bit 1	1x
Eth32AnalogChannel.DI51X1	0x15	Port 3, Bit 5	Port 3, Bit 1	1x
Eth32AnalogChannel.DI61X1	0x16	Port 3, Bit 6	Port 3, Bit 1	1x
Eth32AnalogChannel.DI71X1	0x17	Port 3, Bit 7	Port 3, Bit 1	1x
Eth32AnalogChannel.DI02X1	0x18	Port 3, Bit 0	Port 3, Bit 2	1x
Eth32AnalogChannel.DI12X1	0x19	Port 3, Bit 1	Port 3, Bit 2	1x
Eth32AnalogChannel.DI22X1	0x1A	Port 3, Bit 2	Port 3, Bit 2	1x
Eth32AnalogChannel.DI32X1	0x1B	Port 3, Bit 3	Port 3, Bit 2	1x
Eth32AnalogChannel.DI42X1	0x1C	Port 3, Bit 4	Port 3, Bit 2	1x
Eth32AnalogChannel.DI52X1	0x1D	Port 3, Bit 5	Port 3, Bit 2	1x

Note that the entries above which show both the positive side and negative side with the same input pin can be used for calibration of the differential amplifier. Any nonzero reading from those indicates an offset error within the differential amplifier which you can subtract out of other channels that share the same negative input and gain.

Table 6. Calibration Reference Channels

Constant	Value	Description
Eth32AnalogChannel.R122V	0x1E	Internal 1.22V Voltage Reference
Eth32AnalogChannel.R0V	0x1F	0V (Ground)

The above two entries connect a logical channel to internal chip voltages. They can be used as calibration points to determine errors within the analog conversions.

Example

```
Eth32 dev = new Eth32();

try
{
        // .... Your code that establishes a connection here


        if(dev.AnalogAssignment[0]==Eth32AnalogChannel.SE0)
        {
                // Logical channel 0 is configured for physical
                // single-ended channel 0 (the default)
        }

        // Configure logical channel 7:  Assign it to the
        // difference between bit 4 and bit 1 with 1X gain.
        dev.AnalogAssignment[7]=Eth32AnalogChannel.DI41X1;
}
catch (Eth32Exception e)
{
        // Handle Eth32 errors here
}
```

See Also

AnalogReference Property, AnalogState Property, InputAnalog Method

---

AnalogReference Property

```
Eth32AnalogReference AnalogReference
```

Summary

This property configures the voltage source to be used by the Analog to Digital Converter as the reference voltage for analog conversions. The reference voltage determines the voltage level that will give the highest possible analog reading value. There are three possible voltages that may be used: An externally-generated voltage supplied on the analog reference pin, internal 5V, and internally generated 2.56V.

Parameters

This property does not have any parameters.

Value

This property is a Eth32AnalogReference enumerator type, which has the following valid values:

- Eth32AnalogReference.External - Selects the external, user-supplied voltage.
- Eth32AnalogReference.Internal - Selects the internal 5V source.
- Eth32AnalogReference.Internal256 - Selects the internal 2.56V reference.

Remarks

Note that whatever voltage source is selected will be internally connected to the external voltage reference pin. So for example, if you have a 4V source connected to the external reference pin, you should NOT configure the reference for Internal or Internal256 until you have disconnected the external reference pin.

Also note that if you connect a voltage to the external reference pin, it must not exceed 5V or go below 0V.

See Also

AnalogState Property, InputAnalog Method

AnalogState Property

```
Eth32AnalogState AnalogState
```

Summary

This property enables or disables the Analog to Digital Converter (ADC) portion of the ETH32 device. The ADC must first be enabled before any valid analog readings can be obtained.

Parameters

This property does not have any parameters.

Value

This property is a Eth32AnalogState enumerator type, which has the following valid values:

- Eth32AnalogState.Disabled - The Analog to Digital Converter is disabled. Analog readings will not be valid.

- Eth32AnalogState.Enabled - The Analog to Digital Converter is enabled.

Remarks

Because the analog channels use the same physical pins as digital I/O port 3, enabling the ADC forces port 3 into input mode and sets the output value of port 3 to zero. Changes to the direction register or output value of port 3 are disabled while the ADC remains enabled. Note that regardless of what port 3's direction register and output value were at the time the ADC was enabled, if the ADC is later disabled, port 3 will be left in input mode with an output value of zero.

See Also

InputAnalog Method, AnalogAssignment Property, AnalogReference Property

---

Connect Method

The Connect method is overloaded, with the following options:

```
void Connect(string address)
void Connect(string address, int port)
void Connect(string address, int port, int timeout)
```

Summary

The Connect method is used to open a connection to an ETH32 device. You must call Connect and successfully connect to an ETH32 device before calling other methods or accessing other properties of the Eth32 object. This method does NOT reset the device or change its configuration in any way.

Parameters

- address - The IP address or DNS name of the ETH32 device.
- port - The TCP port to connect to. If an overloaded method without this parameter is called, the constant Eth32.DefaultPort (7152) is used, which is the port the ETH32 listens on.
- timeout - Specifies the maximum time, in milliseconds, that the connection attempt may take, excluding resolving DNS. You may specify a timeout of zero to use the default timeout from the system's TCP/IP stack, which is the behavior for the overloaded methods without this parameter. Note that the method may time out in less time than you specify if the system's timeout is shorter than what you specify. If the method does time out, it will raise an Eth32Exception with ErrorCode of EthError.Timeout

Return Value

This method does not have a return value. If any error occurs, an Eth32Exception will be raised.

Remarks

Once an object is connected to a device, you may not call Connect again on that object unless you first disconnect using the Disconnect Method. Note that your application may have connections open to several ETH32 devices at once. Each requires a separate Eth32 object to be created in your application.

Example

```
Eth32 dev = new Eth32();

try
{
        // Attempt to connect.  If it takes longer than 10 seconds, time out.
        dev.Connect("192.168.1.100", Eth32.DefaultPort, 10000);

        // Now that we're connected, turn on an LED:
        dev.Led[0]=true;
}
catch (Eth32Exception e)
{
        if(e.ErrorCode==EthError.Timeout)
        {
                MessageBox.Show("Timed out while connecting to ETH32.");
        }
        else
        {
                MessageBox.Show("Error connecting to ETH32: " + Eth32.ErrorString(e.ErrorCode));
        }
}
```

See Also

Connected Property, Disconnect Method

Connected Property

```
bool Connected
```

Summary

This is a read-only property that indicates whether the Connect Method has been successfully called on this object and that the Disconnect Method has not been called since then. Reading this property does not cause any communication with the device nor does it verify that the connection to the device is still good. For that, see the VerifyConnection Method.

If there is a connection to the device, this property will read as true. If there is not a connection to the device, rather than raising an exception, this property will simply read false.

Parameters

This property does not have any parameters.

Value

This property is a boolean type. A true value means that the object is connected to an ETH32 device, while false means that it is not.

Example

```
Eth32 dev = new Eth32();

try
{
        // .... Your code here

        // Assume that we don't know for sure whether the dev object
        // is connected to a device, but that if it is, we want to
        // disconnect it.  This code accomplishes that:
        if(dev.Connected)
        {
                dev.Disconnect();
        }
}
catch (Eth32Exception e)
{
        // Handle Eth32 errors here
}
```

See Also

Connect Method, Disconnect Method, VerifyConnection Method

ConnectionFlags Method

```
Eth32ConnectionFlag ConnectionFlags(int reset)
```

Summary

The ETH32 device maintains several flag bits for each individual active TCP/IP connection. The flags indicate conditions that are (or were) present for that connection. Currently, these flags are used to indicate whether any data that needed to be sent to your application from the ETH32 device had to be discarded due to lack of queue space. This method retrieves the flags for this connection to the device. If instructed to do so, the method also clears all of the flags for this connection to zero immediately after retrieving them.

Parameters

- reset - If nonzero, specifies that the flags for this connection should be reset to zero immediately after retrieving them.

Return Value

This method returns a Eth32ConnectionFlag enumerator type. The return value may be made up of any combination (that is, a bitwise or) of the following enumerator flags. Each flag indicates which kind of data had to be discarded due to a full queue.

- Eth32ConnectionFlag.None - If the return value equals this exactly, then no flags were set.
- Eth32ConnectionFlag.Response - Response to a query for information (for example InputByte Method).
- Eth32ConnectionFlag.DigitalEvent - Digital event notification.
- Eth32ConnectionFlag.AnalogEvent - Analog event notification.
- Eth32ConnectionFlag.CounterEvent - Counter event (rollover or threshold) notification.

Remarks

To understand the role of the connection flags, consider the following example. Suppose that digital events are enabled on port 0, bit 0 for your connection to the ETH32. Now suppose that port 0, bit 0 begins pulsing rapidly, generating a steady stream of event notifications. Finally, suppose that the connection to your application is having trouble (losing packets, etc). Due to the nature of TCP/IP, the event notifications must be retained in the queue of the ETH32 device until a TCP/IP acknowledgement for them has been received from the PC (in case they need to be retransmitted). If the TCP/IP acknowledgements do not come promptly and the events keep occurring, the queue will eventually fill up and the ETH32 device will be forced to simply discard any new event notifications. Although this scenario is undesirable and should be avoided, if it does happen, it is helpful for your application to be able to detect that it happened and that data was lost. The flags keep track of this individually for each TCP/IP connection (that is, a full queue on one connection will not affect flags on another). Note that the flags are informational only - they do not affect the behavior of the device.

Once a flag is set, it will remain set until it is reset back to zero by passing a nonzero number to the *reset* parameter of this method. In this case, the flags will only be reset to zero if the connection has enough space to queue up the reply data. In other words, the flags will not be lost if the response itself is unable to be queued.

The connection flags for new connections always start out as zero. When the ResetDevice Method is called, the flags for the connection it was received on are cleared, but the flags for any other active connections are not affected.

Example

```
Eth32 dev = new Eth32();
Eth32ConnectionFlag flags;

try
{
        // .... Your code that establishes a connection here

        // Retrieve the connection flags for this connection and
        // simultaneously clear them to zero.
        flags=dev.ConnectionFlags(1);

        // See which flags are set
        if((flags & Eth32ConnectionFlag.Response)==Eth32ConnectionFlag.Response)
        {
                // The device ran out of queue space at some point
                // when it was trying to respond to a query for information.
        }

        if((flags & Eth32ConnectionFlag.DigitalEvent)==Eth32ConnectionFlag.DigitalEvent)
        {
                // Some digital event data was lost due to running out
                // of queue space.
        }

        // and so on


        // Or, to check whether any flags at all are set:
        if(flags == Eth32ConnectionFlag.None)
        {
                // No flags whatsoever are set
        }
        else
        {
                // At least one flag is set
        }
}
catch (Eth32Exception e)
{
        // Handle Eth32 errors here
}
```

See Also

VerifyConnection Method

CounterRollover Property

```
int CounterRollover[int counter]
```

Summary

This property defines the maximum permissible value for a counter. After the counter reaches the rollover value, the next count will cause the counter to be reset to 0 and a rollover event notification will be sent to any connections that have enabled that rollover event. For example, with a rollover threshold set to 35, the counter value will progress as follows: ..., 33, 34, 35, 0, 1, ... Because the comparisons and reset are done directly in hardware, no counts are missed during a rollover.

The valid range of the rollover threshold is from 0 to the maximum value of the counter (65535 for 16-bit counter 0, and 255 for 8-bit counter 1). The powerup default rollover threshold is 255 for 8-bit and 65535 for 16-bit counters.

Parameters

- counter - Specifies the counter number (0 or 1).

Value

This property is an integer. For counter 0 (a 16-bit counter), this may range from 0-65535. For counter 1 (an 8-bit counter), this may range from 0-255.

Remarks

There is one special case involving rollover thresholds. When the counter value is manually set to exactly the threshold value by writing to the CounterValue Property, the rollover will NOT occur and the rollover event will NOT fire on the next counter increment. Instead, the counter will increment past the threshold value. The event will not fire until the counter value has wrapped around and again exceeds the threshold. For example, suppose the rollover threshold is set to 10 on an 8-bit counter and the CounterValue Property is used to set the counter value to 10. As the input line pulses, the counter value would increment as follows: 11, 12, ..., 255, 0, 1, ..., 10, 0, 1, ..., 10, 0, ...

Please note that defining a rollover threshold with this property does not enable the current connection to actually receive the rollover event notifications when they occur. These must be enabled separately using the EnableEvent Method. Also note that rollover thresholds are common to all connections. Changing the thresholds will affect other connections if they are utilizing that particular counter.

See Also

CounterState Property, CounterThreshold Property, EnableEvent Method

---

CounterState Property

```
Eth32CounterState CounterState[int counter]
```

Summary

This property allows you to control or retrieve the state of the two counters on the ETH32 device. The counter state configures which input signal edge (rising or falling) will increment the counter value or whether the counter is disabled. Setting or accessing this property does not affect the current counter value in any way. For example, a counter that is disabled and then enabled again will retain its value.

Parameters

- int counter - Specifies the counter number (0 or 1).

Value

This property is a Eth32CounterState enumerator type, which has the following valid values:

- Eth32CounterState.Disabled - The counter is disabled. The counter value may still be accessed, but the counter will not increment as a result of input signals.
- Eth32CounterState.Falling - The counter will increment on the falling edge of the input signal.
- Eth32CounterState.Rising - The counter will increment on the rising edge of the input signal.

See Also

CounterRollover Property, CounterValue Property

CounterThreshold Property

```
int CounterThreshold[int counter]
```

Summary

This property defines a counter event threshold that will cause an event to fire as the counter value passes the threshold. On the ETH32 device, only Counter 0 supports this (although both counters support rollover thresholds). An event is fired as a result of the counter surpassing the threshold, not meeting it. For example, with a threshold of 9, the counter's value would increment from 8 to 9 without firing the event, but it would fire as the counter incremented from 9 to 10. The valid range for a counter event threshold is from 0 to the maximum possible counter value (65535 for 16-bit counter 0). The powerup default threshold is 0. The threshold has no other side-effects on the counter - it does not reset the counter to 0 like the rollover threshold.

Parameters

- counter - Specifies the counter number. This must be 0.

Value

This property is an integer. The valid range is from 0 to the maximum possible counter value (65535 for 16-bit counter 0).

Remarks

Please note that defining a threshold with this property does not enable the current connection to actually receive the event notifications when they occur. These must be enabled separately using the EnableEvent Method. Also note that event thresholds are common to all connections. Changing the thresholds will affect other connections if they are utilizing that particular counter event.

See Also

CounterState Property, CounterRollover Property, EnableEvent Method

---

CounterValue Property

```
int CounterValue[int counter]
```

Summary

This property allows you to read or write the current value of the counters on the ETH32 device. After you have enabled the counter with the CounterState Property, the value of the counter indicates how many times the counter has been incremented by the external counter input. This property can also be written in order to set the counter value, which can be useful for initializing the counter. All counters begin with a value of zero after powerup or reset.

Parameters

- counter - Specifies the counter number (0 or 1).

Value

This property is an integer. For counter 0 (a 16-bit counter), this may range from 0-65535. For counter 1 (an 8-bit counter), this may range from 0-255.

See Also

CounterState Property, CounterRollover Property

DisableEvent Method

```
void DisableEvent(Eth32EventType eventtype, int port, int bit)
```

Summary

This method instructs the ETH32 device to stop sending event notifications for the specified event on this connection to the device. It performs the opposite task of the EnableEvent Method.

Parameters

- eventtype - The type of event to disable. This parameter is a Eth32EventType enumerator type, which has the following valid values:
  - Eth32EventType.Digital - Digital I/O event. This includes port events and bit events.
  - Eth32EventType.Analog - Analog event based on thresholds defined with the SetAnalogEventDef Method.
  - Eth32EventType.CounterRollover - Counter rollover event, which occurs when the counter rolls over to zero.
  - Eth32EventType.CounterThreshold - Counter threshold event, which occurs when the counter passes a threshold defined with the CounterThreshold Property.
  - Eth32EventType.Heartbeat - Periodic event sent by the device to indicate the TCP/IP connection is still good.
- port - For digital events, specifies the port number, for analog events, specifies the bank number, and for either counter event, specifies the counter number.
- bit - For digital events, this should be -1 for port events or the bit number (0-7) for bit events. For analog events, this specifies the analog channel number (0-7).

Return Value

This method does not return a value.

See Also

EnableEvent Method

Disconnect Method

```
void Disconnect()
```

Summary

This method closes the connection to the ETH32 device and cleans up all of the resources within the API that were used for the connection. After this method returns, most of the methods and properties of the object won't be able to be successfully used until another connection has been formed using the Connect Method.

Parameters

This method does not have any parameters.

Return Value

This method does not return a value.

Remarks

You should be careful to always call this method when you are finished using the device. The device has a limited number of connections it can support and if you do not disconnect and your application continues executing, you will continue using one of those connections. If you fail to call this method, your connections will remain open potentially until your application terminates.

In this .NET class, an object's connection is automatically closed at the time of garbage collection if it is not already disconnected. However, you should never depend on .NET garbage collection to do this because .NET garbage collection is non-deterministic. This means that garbage collection may occur at a much later time than when you cease using an object. Depending on the memory usage of your application, garbage collection may not occur until your application terminates.

See Also

Connect Method, Connected Property

EmptyEventQueue Method

```
void EmptyEventQueue()
```

Summary

This method empties the event queue within the API. This method does not have an effect on the ETH32 device itself.

Parameters

This method does not have any parameters.

Return Value

This method does not return a value.

See Also

EventQueueCurrentSize Property, EventQueueLimit Property

EnableEvent Method

```
void EnableEvent(Eth32EventType eventtype, int port, int bit, int id)
```

Summary

This method enables reception of the specified event on this connection to the device. The ETH32 device only sends event notifications to those connections that specifically request them, so this method requests notification for the specified event from the device, as well as internally assigns the event an ID number provided by you.

Parameters

- eventtype - The type of event to enable. This parameter is a Eth32EventType enumerator type, which has the following valid values:
  - Eth32EventType.Digital - Digital I/O event. This includes port events and bit events.
  - Eth32EventType.Analog - Analog event based on thresholds defined with the SetAnalogEventDef Method.
  - Eth32EventType.CounterRollover - Counter rollover event, which occurs when the counter rolls over to zero.
  - Eth32EventType.CounterThreshold - Counter threshold event, which occurs when the counter passes a threshold defined with the CounterThreshold Property.
  - Eth32EventType.Heartbeat - Periodic event sent by the device to indicate the TCP/IP connection is still good.
- port - For digital events, specifies the port number, for analog events, specifies the bank number, and for either counter event, specifies the counter number.
- bit - For digital events, this should be -1 for port events or the bit number (0-7) for bit events. For analog events, this specifies the analog channel number (0-7).
- id - You may specify any number to be associated with this event.

Return Value

This method does not return a value.

Remarks

The *id* parameter allows you to assign any arbitrary number to this particular event. The ID you assign is included with the event information whenever this event fires. The idea is that you can identify a particular event with a single comparison rather than needing to inspect several pieces of data such as the event type, port number, and bit number. The ID number is completely arbitrary and multiple events may be given the same ID number if desired. The ID numbers are stored within the API and are not sent to the ETH32

device.

One other minor technicality is that the heartbeat event is permanently enabled on the ETH32 device itself for purposes of connection maintenance. Therefore, for the heartbeat event, this method simply enables the event within the API, meaning that when the event comes in, rather than being discarded it will be added to the event queue. The one small side-effect to this fact is that if you have enabled reception of the heartbeat event and another connection calls the ResetDevice Method, you will continue to receive heartbeat events, whereas all other event types will have been disabled on the device itself. Note that if you call ResetDevice on your own connection, it automatically disables the heartbeat event within the API for your connection, so in that case it is not an issue.

Example

```
// This example is a very simple, yet complete (that is, compilable) example
// of how to utilize events
using System;
using WinfordEthIO;

public class MyExample
{
        public static void MyEth32EventHandler(Object s, EventArgs e)
        {
                // This is our event handler function.  This function
                // will be called every time an event notification arrives
                // from the device.  See below in the Main() function for
                // how this function gets "registered" as the event handler.
                Eth32 sender;
                Eth32EventArgs args;

                sender = s as Eth32;
                if(sender == null)
                {
                        // sender wasn't really an Eth32 object - quit
                        return;
                }

                args = e as Eth32EventArgs;
                if(args == null)
                {
                        // The arguments weren't really Eth32EventArgs - quit
                        return;
                }

                System.Windows.Forms.MessageBox.Show("An event has fired.  ID: "
                      + args.event_info.id + " Value: " + args.event_info.val);
        }

        static void Main()
        {
                try
                {
                        Eth32 dev;

                        dev = new Eth32();

                        // NOTE: Substitute the IP address or DNS name of your device here.
                        dev.Connect("192.168.1.100");

                        // Register our event handler function to handle all incoming events
                        dev.HardwareEvent += new EventHandler(MyEth32EventHandler);

                        // If there is a pushbutton connected between Port 0, bit 0 and ground,
                        // then we can provide an internal pullup causing it to float high by
```

```
                        // doing:
                        dev.OutputBit(0, 0, 1);

                        // Look for events on Port 0, bit 0.
                        dev.EnableEvent(Eth32EventType.Digital, 0, 0, 100);


                        // Display a MessageBox.  This function will not return until the
                        // user clicks OK, so it will keep the application running until then.
                        System.Windows.Forms.MessageBox.Show("When you're finished monitoring events, "
                              + "click OK to end application.");
                }
                catch (Eth32Exception e)
                {
                        // Handle Eth32 errors here
                        System.Windows.Forms.MessageBox.Show("Eth32 exception: " + e.ToString());
                }


        }
}
```

See Also

Event Handler, DisableEvent Method

ErrorString Method

```
static string ErrorString(EthError errorcode)
```

Summary

This method translates an error code into a string which briefly describes the error.

Parameters

- errorcode - The error code to translate into a string. This parameter is a EthError enumerator type. Possible error codes are listed in the Error Codes section.

Return Value

This method returns a string, which provides a brief description of the given error code.

Remarks

This is a static method, meaning that it is not called on a specific object instance, but directly on the Eth32 class. This of course means that this method can be called at any time. It is not necessary to have a connection to an ETH32 device, nor is it even necessary to have an instance of the Eth32 class.

Example

```
Eth32 dev = new Eth32();

try
{
        // .... Your code that establishes a connection here
```

```
        // .... More of your code that performs operations on the device or other things.
}
catch (Eth32Exception e)
{
        MessageBox.Show("ETH32 Error: " + Eth32.ErrorString(e.ErrorCode));
}
```

See Also

Error Handling Section

EventQueueCurrentSize Property

```
int EventQueueCurrentSize
```

Summary

This read-only property allows you to determine how many events are currently in the event queue within the API. This property does not communicate with the ETH32 device or provide information about the device itself. For more information about the API event queue, see the EventQueueLimit Property.

Parameters

This property does not have any parameters.

Value

This property is an integer. The value of the property is the number of events currently waiting in the API's event queue.

See Also

EmptyEventQueue Method, EventQueueLimit Property

EventQueueLimit Property

```
int EventQueueLimit
```

Summary

This property controls the maximum allowable size of the event queue within the API. If a nonzero maximum size is configured for the event queue (which is the default when a new connection is created), the API will enable events and queue any events that arrive while your event handler function is already busy processing an event. If a zero maximum size is configured, event processing will be disabled. This property only controls the behavior of the API. It does not affect anything on the actual ETH32 device.

Parameters

This property does not have any parameters.

Value

This property is an integer. Its value specifies the maximum number of events that are allowed to be queued by the API.

Remarks

Your event handler routine is called once for each event notification that is sent by the device. Events are processed one at a time and in the sequence that they are sent by the device. The event queue is used to store events that have arrived, but have not yet been sent to your event handler routine. This is particularly important if your event handler routine takes a significant time to execute.

If the event queue ever becomes full and more events arrive, the behavior of the API will depend on the current setting of the EventQueueMode Property.

Example

```
Eth32 dev = new Eth32();

try
{
        // .... Your code that establishes a connection here

        // Configure the event queue to hold up to 1,000 events.
        // If the queue is ever full and more events arrive, discard
        // the new events.
        dev.EventQueueLimit=1000;
        dev.EventQueueMode=Eth32QueueMode.DiscardNew;

}
catch (Eth32Exception e)
{
        // Handle Eth32 errors here
}
```

See Also

EnableEvent Method, EventQueueCurrentSize Property, EventQueueMode Property

---

EventQueueMode Property

```
Eth32QueueMode EventQueueMode
```

Summary

This property configures the behavior of the event queue within the API. If the event queue ever becomes full (reaches the limit configured by the EventQueueLimit Property) and new events arrive, either old events will be shifted out to make room for the new, or the new events will be ignored, depending on the behavior you have specified with this property. The Eth32QueueMode.DiscardNew setting is the default when a new connection is created.

Parameters

This property does not have any parameters.

Value

This property is a Eth32QueueMode enumerator type, which has the following valid values:

- Eth32QueueMode.DiscardNew - When the queue is full, discard any new events.
- Eth32QueueMode.DiscardOld - When the queue is full, shift out the oldest event to make room for the new event at the end of the queue.

Remarks

The event queue size that is considered full is defined by the EventQueueLimit Property.

See Also

EventQueueLimit Property

FirmwareMajor Property

```
int FirmwareMajor
```

Summary

This read-only property retrieves the "major" portion of the firmware version number from the device. The firmware version consists of a major number and minor number. When displayed as a string, it is typically formatted as major.minor with minor zero-padded to three digits if necessary. For example, for release 2.001, the major number is 2 and the minor number is 1.

Parameters

This property does not have any parameters.

Value

This property is an integer. Its value is the major number of the firmware version.

See Also

FirmwareMinor Property

FirmwareMinor Property

```
int FirmwareMinor
```

Summary

This read-only property retrieves the "minor" portion of the firmware version number from the device. The firmware version consists of a major number and minor number. When displayed as a string, it is typically formatted as major.minor with minor zero-padded to three digits if necessary. For example, for release 2.001, the major number is 2 and the minor number is 1.

Parameters

This property does not have any parameters.

Value

This property is an integer. Its value is the minor number of the firmware version.

See Also

FirmwareMajor Property

GetAnalogEventDef Method

```
void GetAnalogEventDef(int bank, int channel, out int lomark, out int himark)
```

Summary

This method retrieves the low and high thresholds defined for the specified analog event bank and channel. Please see the SetAnalogEventDef Method for more information about the analog event definition and thresholds.

Parameters

- bank - Identifies which bank of analog events from which to retrieve information (0 or 1).
- channel - Identifies the analog channel (0-7).

- lomark - Output parameter which will receive the low threshold (8-bit value) for the analog event.
- himark - Output parameter which will receive the high threshold (8-bit value) for the analog event.

Return Value

This method does not return a value.

Remarks

Note that this method does not retrieve the default value that was specified when the thresholds were set. This is because the default value is only used during the moment that the thresholds are defined and is not applicable after that point.

See Also

SetAnalogEventDef Method

GetDirection Method

```
int GetDirection(int port)
```

Summary

This method retrieves the current direction register for the specified digital I/O port. See the SetDirection Method for further description of the direction register.

Parameters

- port - The port number (0-5).

Return Value

This method returns an integer. The return value is the port's current direction register.

See Also

GetDirectionBit Method, SetDirection Method, SetDirectionBit Method

---

GetDirectionBit Method

```
int GetDirectionBit(int port, int bit)
```

Summary

This method retrieves the value of a single bit of a port's direction register. It is provided simply for convenience, since it internally calls the GetDirection Method to determine the value of the specified bit.

Parameters

- port - Specifies the port number (0-5).
- bit - Specifies the bit number (0-7) within the port.

Return Value

This method returns an integer. The return value is the value of the specified direction bit of the specified port. Zero indicates the bit is configured for input and nonzero indicates it is configured for output.

See Also

GetDirection Method, SetDirection Method, SetDirectionBit Method

---

GetDirectionBitBool Method

```
bool GetDirectionBitBool(int port, int bit)
```

Summary

This method is the same as the GetDirectionBit Method but returning a boolean value where true indicates a set bit (1), meaning it is configured for output, and false indicates a cleared bit (0), meaning it is configured for input. It is simply provided for convenience.

GetDirectionByte Method

```
byte GetDirectionByte(int port)
```

Summary

This method is the same as the GetDirection Method but returning a byte. It is simply provided for convenience.

GetEeprom Method

```
byte[] GetEeprom(int address, int length)
```

Summary

This method retrieves data from the non-volatile EEPROM memory of the device.

Parameters

- address - The starting location from which data should be retrieved (0-255).

- length - The number of bytes to retrieve. Valid values for this parameter depend on what is provided for the address parameter. For example, with an address of 0, you may specify a length of all 256 bytes, but with an address of 255, length may only be 1.

Return Value

This method returns a byte array containing the requested data.

See Also

SetEeprom Method

GetPwmParameters Method

```
void GetPwmParameters(int channel, out Eth32PwmChannel state, out double freq, out double duty)
```

Summary

This method is provided for your convenience in working with all of the various PWM settings. It internally calls several of the other API functions to determine the current state of the specified PWM channel and calculate its configuration in more familiar terms (hertz and percentage). This method calculates the frequency and duty cycle of the channel from the PWM base period and the channel's duty period.

Parameters

- channel - Specifies the PWM channel number (0 or 1).
- state - Output parameter which will receive the current state of the PWM channel. This will be one of the following values of the Eth32PwmChannel enumerator:
  - Eth32PwmChannel.Disabled - The PWM pin is configured as a normal digital I/O pin.
  - Eth32PwmChannel.Normal - The PWM pin is configured as a PWM output. It will be high for the time specified by the duty period and low for the rest of the PWM base period.
  - Eth32PwmChannel.Inverted - The PWM pin is configured as a PWM output. It will be low for the time specified by the duty period and high for the rest of the PWM base period.
- freq - Output parameter which will receive the current frequency of the PWM channels in Hertz.
- duty - Output parameter which will receive the duty cycle of the PWM channel. This may range from 0.00 to 1.00, representing the duty cycle as a percentage.

Return Value

This method does not return a value.

See Also

SetPwmParameters Method

InputAnalog Method

```
int InputAnalog(int channel)
```

Summary

This method retrieves an analog reading from one of the analog channels on the device. The analog readings are only meaningful when the ADC has been enabled (see the AnalogState Property). The analog readings are 10-bit values. See below for further explanation of their meaning.

Parameters

- channel - Specifies the logical analog channel (0-7) to read. Note that each logical analog channel may be arbitrarily assigned to physical channels using the AnalogAssignment Property.

Return Value

This method returns an integer. The return value is the reading from the specified channel.

Remarks

The reading that is obtained with this method is a 10-bit value (range of 0-1023) representing the voltage level relative to the analog reference voltage. The exact interpretation depends on whether a single-ended or differential channel has been selected (see the AnalogAssignment Property).

For single-ended channels, the reading is:

```
(analog reading) = (channel voltage * 1024) / (voltage reference)
```

For example, a reading of 0 means 0V and a reading of 1023 means a voltage just under the voltage reference (assuming internal 5V reference, about 4.99V). Once you have the analog reading, you can calculate the input voltage that produced it by calculating:

```
voltage = (analog reading)/1024 * (voltage reference)
```

For differential channels, the reading is:

```
(analog reading) = 512 + (positive side voltage - negative side voltage) * GAIN * 512 / (voltage reference)
```

For example, assuming a gain of 1X, a reading of 0 means the positive pin is (voltage reference) volts less than the negative pin, a reading of 512 means the positive pin and negative pin are at the same voltage, and a reading of 1023 means the positive pin is almost (voltage reference) volts higher than the negative pin. Once you have the analog reading, you can calculate the voltage of the positive pin relative to the negative pin by calculating:

```
voltage = (analog reading - 512) / 512 * (voltage reference)
```

Example

```
Eth32 dev = new Eth32();
int chan0;
double voltage;

try
{
        // .... Your code that establishes a connection here


        // Enable the Analog to Digital Converter
        dev.AnalogState=Eth32AnalogState.Enabled;

        // Configure logical channel 0 to read the physical channel 0 relative to ground (single-ended)
        // This is the power-on default anyway, but is shown here for completeness:
        dev.AnalogAssignment[0]=Eth32AnalogChannel.SE0;

        // Configure the analog voltage reference to be the internal 5V source
        dev.AnalogReference=Eth32AnalogReference.Internal;

        // Finally, read the voltage on channel 0
        chan0=dev.InputAnalog(0);

        // Now, determine whether the voltage was >= 3V.  Remember
        // we're using a 5V voltage reference.
        if( chan0 >= (3.0/5.0 * 1024) )
        {
                // The voltage on channel 0 was at least 3V
        }
        else
        {
                // The voltage was less than 3V
        }

        // If you want to calculate the voltage:
        voltage = chan0 / 1024.0 * 5.0;
}
catch (Eth32Exception e)
{
        // Handle Eth32 errors here
}
```

See Also

AnalogAssignment Property, AnalogReference Property, AnalogState Property, InputAnalogUShort Method

InputAnalogUShort Method

```
ushort InputAnalogUShort(int channel)
```

Summary

This method is the same as the InputAnalog Method but returning an unsigned short. It is simply provided for convenience.

InputBit Method

```
int InputBit(int port, int bit)
```

Summary

This method retrieves the value of a single bit within a digital I/O port. It is provided simply for convenience, since it internally calls the InputByte Method to determine the value of the specified bit.

Parameters

- port - Specifies the port number (0-5) to read.
- bit - Specifies the bit number (0-7).

Return Value

This method returns an integer. The return value is the current value (0 or 1) of the specified bit.

See Also

InputByte Method, OutputBit Method, SetDirectionBit Method, InputBitBool Method

---

InputBitBool Method

```
bool InputBitBool(int port, int bit)
```

Summary

This method is the same as the InputBit Method but returning a boolean, where true indicates a high bit (1) and false indicates a low bit (0). It is simply provided for convenience.

InputByte Method

```
int InputByte(int port)
```

Summary

This method retrieves the current input value of the specified digital I/O port on the device. When a port is configured as an input port (using the SetDirection Method), the input value represents the voltage levels on the port's pins. For each bit, a low voltage (close to 0V) yields a 0-bit in the input value and a high voltage (close to 5V) yields a 1-bit.

Parameters

- port - Specifies the port number (0-5) to read.

Return Value

This method returns an integer. The return value is the current input value of the specified port.

Example

```
Eth32 dev = new Eth32();
int portval;

try
{
        // .... Your code that establishes a connection here

        // Read the input value of port 2
        portval=dev.InputByte(2);

        // See whether any of bits 0-3 are high (1)
        if( (portval & 0x0F) != 0 )
        {
                // At least one of bits 0-3 are high
        }
        else
        {
                // None of bits 0-3 are high
        }

}
catch (Eth32Exception e)
{
        // Handle Eth32 errors here
}
```

See Also

InputBit Method, InputByteByte Method, OutputByte Method, SetDirection Method

InputByteByte Method

```
byte InputByteByte(int port)
```

Summary

This method is the same as the InputByte Method but returning a byte. It is simply provided for convenience.

InputSuccessive Method

```
InputSuccessive(int port, int maxcount, out int status)
```

Summary

This method instructs the ETH32 device to read the specified port multiple times in succession until two consecutive reads yield the same result. This method is useful for situations where a multi-bit value is being read, for example, the output of a digital counter chip. When reading such a value, it is always possible to read the value during a transition state as bits are changing and an invalid value is represented. By requiring that two successive reads match, any invalid transition values are automatically ignored. The device continues to read the port until one of the following conditions is met:

- Two successive (in other words, back to back) reads give the same port value. This value is returned.
- The port was read the maximum number of times specified in the command without a match occurring.

This functionality is implemented directly within the ETH32 device (as opposed to the API), making it very fast and efficient with network traffic.

Parameters

- port - Specifies the port number (0-3) to read.
- max - The maximum number of times to read the port (2-255).
- status - Output parameter which will receive the number of times the port had to be read to get a successive match. If no match was ever seen, this will be zero.

Return Value

This method returns an integer. The return value is the last value read from the port, regardless of whether or not two successive reads ever matched.

Example

```
Eth32 dev = new Eth32();
int portval;
int status;

try
{
        // .... Your code that establishes a connection here

        // Read the value of an 8-bit counter on port 0, limit to 20 reads
        portval=dev.InputSuccessive(0, 20, status);

        if(status==0)
        {
                // Never saw the same value twice in a row
        }
        else
        {
                // The port value is in the portval variable
        }

}
catch (Eth32Exception e)
{
        // Handle Eth32 errors here
}
```

See Also

InputByte Method, SetDirection Method, InputSuccessiveByte Method

---

InputSuccessiveByte Method

```
byte InputSuccessiveByte(int port, int maxcount, out int status)
```

Summary

This method is the same as the InputSuccessive Method but returning a byte. It is simply provided for convenience.

Led Property

```
bool Led[int lednum]
```

Summary

This property allows you to control or retrieve the state of the two LED’s built into the ETH32 device.

Parameters

- lednum - Identifies the LED (0 or 1) to control or inspect.

Value

This property is a boolean type. A true value means the LED is on and a false value means the LED is off.

Example

```
Eth32 dev = new Eth32();

try
{
        // .... Your code that establishes a connection here

        // Determine whether LED 0 is on or off
        if(dev.Led[0])
        {
                // LED is on
        }
        else
        {
                // LED is off
        }

        // Turn on LED 1
        dev.Led[1]=true;
}
catch (Eth32Exception e)
{
        // Handle Eth32 errors here
}
```

OutputBit Method

```
void OutputBit(int port, int bit, int val)
void OutputBit(int port, int bit, bool val)
```

Summary

This overloaded method alters a single bit of the output value of any I/O port without affecting the value of any other bits. See the OutputByte Method for further description of the output value.

Parameters

- port - The port number (0-5).
- bit - The bit number (0-7).

- val - Any nonzero number or true sets the bit to 1 and zero or false clears the bit to 0.

Return Value

This method does not return a value.

Remarks

This method alters the specified bit's value in a single operation directly on the ETH32 device. In other words, it does NOT read the current value over the network, modify it and then write it back. By doing it in a single operation, this avoids the potential of inadvertently overwriting changes made to other bits by other connections.

Port 3 shares its pins with the analog channels. When the ADC is enabled, all pins of port 3 are forced into input mode and the output value is set to zero. Port 3's output value cannot be modified while the ADC is enabled.

See Also

InputBit Method, OutputByte Method, SetDirectionBit Method

OutputByte Method

```
void OutputByte(int port, int val)
```

Summary

This method writes a new output value to one of the digital I/O ports on the device. When the port is configured as an output port (using the SetDirection Method), each bit of the output value determines the voltage (0 or 5V) of the corresponding bit of the port. When the port is configured as an input port, any 1-bits in the output value enables a weak pullup for that bit of the port.

Parameters

- port - The port number to write to (0-5).
- value - The new value for the port. This may be 0-255 for ports 0-3 and 0-1 for the single-bit ports 4 and 5.

Return Value

This method does not return a value.

Remarks

Port 3 shares its pins with the analog channels. When the ADC is enabled, all pins of port 3 are forced into input mode and the output value is set to zero. Port 3's output value cannot be modified while the ADC is enabled.

Example

```
Eth32 dev = new Eth32();

try
{
        // .... Your code that establishes a connection here

        // Set port 0 pins to be outputs
        dev.SetDirection(0, 255);

        // Write a new value for port 0
        dev.OutputByte(0, 85);
}
catch (Eth32Exception e)
{
        // Handle Eth32 errors here
}
```

See Also

InputByte Method, OutputBit Method, Readback Method, SetDirection Method

---

ProductID Property

```
int ProductID
```

Summary

This read-only property retrieves the product ID from the device, which identifies the type/model of the device.

Parameters

This property does not have any parameters.

Value

This property is an integer. Its value is a numeric code representing the type or model of the device.

See Also

SerialNum Property

PulseBit Method

```
void PulseBit(int port, int bit, Eth32PulseEdge edge, int count)
```

Summary

This method outputs a burst of pulses on the port and bit specified. This can be useful, for example, in quickly clocking an external digital counter a specified number of times. You should ensure that the specified bit is configured as an output bit before calling this method. The ETH32 device implements the pulse functionality (as opposed to the API), which means it is performed very quickly and is efficient for network traffic.

Parameters

- port - The port number (0-5).
- bit - The bit number (0-7) on the specified port that should be pulsed.
- edge - Specifies whether the pulses should be falling or rising edge. This parameter is a Eth32PulseEdge enumerator type, which has the following valid values:
  - Eth32PulseEdge.Falling - Bit is set low, then high, for each pulse.
  - Eth32PulseEdge.Rising - Bit is set high, then low, for each pulse.
- count - The number of times to pulse the bit. May be up to 255.

Return Value

This method does not return a value.

Remarks

The falling edge mode would typically be used on a bit that is initially high (and likewise rising edge with low), but this is not required. If a single falling edge pulse is performed on a bit that is already low, the pulse will end up simply setting the bit high. The reverse applies to a rising edge pulse where the bit is already high.

See Also

OutputBit Method, SetDirectionBit Method

PwmBasePeriod Property

```
int PwmBasePeriod
```

Summary

This property configures the main PWM clock to have a cycle period of the given number of counts. This defines the base frequency that will be used for the PWM channels. The base frequency is not individually selectable for each channel, so this property affects both PWM outputs. Each complete PWM waveform will have a duration of (BasePeriod + 1) PWM clock cycles. The PWM clock counts at a rate of 2 MHZ. This means, for example, that specifying a period of 99 would result in a frequency of 20 KHZ

(2,000,000/(99+1)). The base period is specified as a 16-bit number that may range from a value of 49 (40 KHZ) to a value of 65,535 (30.5 HZ).

Parameters

This property does not have any parameters.

Value

This property is an integer. It specifies the number of PWM clock counts that make up the base period of the PWM channels. This may range from 49 - 65535.

See Also

PwmChannel Property, PwmClockState Property, PwmDutyPeriod Property, SetPwmParameters Method

---

PwmChannel Property

```
Eth32PwmChannel PwmChannel[int channel]
```

Summary

This property configures the state of the PWM channels. When a channel is disabled, the I/O pin will function as a normal digital I/O pin. When the channel is enabled, that I/O pin will be overridden and the pin will become the PWM output. However, note that the pin must be put into output mode using the SetDirection Method or SetDirectionBit Method.

Parameters

- channel - Specifies the PWM channel number whose state should be set (0 or 1).

Value

This property is a Eth32PwmChannel enumerator type, which has the following valid values:

- Eth32PwmChannel.Disabled - The PWM pin will function as a normal digital I/O pin.
- Eth32PwmChannel.Normal - The PWM pin will function as a PWM output. It will be high for the time specified by the duty period and low for the rest of the PWM base period.
- Eth32PwmChannel.Inverted - The PWM pin will function as a PWM output. It will be low for the time specified by the duty period and high for the rest of the PWM base period.

See Also

PwmBasePeriod Property, PwmClockState Property, PwmDutyPeriod Property, SetPwmParameters Method

PwmClockState Property

```
Eth32PwmClock PwmClockState
```

Summary

This property enables or disables the PWM clock from counting. The PWM clock is shared between both PWM outputs of the device. When the PWM clock is disabled, the PWM outputs will be idle (not pulsing). The PWM clock may be enabled or disabled independently of whether the individual PWM channel outputs are enabled or disabled.

Parameters

This property does not have any parameters.

Value

This property is a Eth32PwmClock enumerator type, which has the following valid values:

- Eth32PwmClock.Disabled - Disables the PWM clock.
- Eth32PwmClock.Enabled - Enables the PWM clock.

See Also

PwmBasePeriod Property, PwmChannel Property, PwmDutyPeriod Property, SetPwmParameters Method

PwmDutyPeriod Property

```
int PwmDutyPeriod[int channel]
```

Summary

This property defines the duty period for a PWM channel, which is the length of time the PWM output is active during each PWM cycle. The duty period is specified as PWM clock counts less one. In other words, when the PWM channel state is in normal mode, the PWM output will be high for (DutyPeriod + 1) counts of the PWM clock and low for the remainder of the clock counts in the cycle. The length of the PWM cycle is called the base period and set using the PwmBasePeriod Property.

Parameters

- channel - Specifies the PWM channel number (0 or 1).

Value

This property is an integer. The value specifies the duty period for the channel in terms of PWM clock counts. The valid range is from 0-65535.

Remarks

Remember that the base period (set with the PwmBasePeriod Property) is shared between both PWM channels on the device. However, the duty period (set with this property) is individually configurable for each channel. The recommended approach is to choose a PWM frequency that is appropriate for both channels and set the base period accordingly once during initialization. After that point, the individual duty periods for each channel should be set whenever necessary in order to alter the percentage of time the channel is on (duty cycle).

Any 16-bit value can be specified for the period, from 0 to 65535. Note that if a duty period is given that is greater than or equal to the current PWM base period, the output will be a solid high (in normal mode) or a solid low (in inverted mode). If a duty period of 0 is given, the output will not be solid, but rather it will have a short spike during each period of the PWM clock.

Example

```
Eth32 dev = new Eth32();

try
{
        // .... Your code that establishes a connection here

        // Set up PWM channel 0 to have a 10 KHZ, 60% PWM signal:

        // First, set up the base period to give a frequency of 10 KHZ
        // This is calculated as:
        //   (2,000,000)/(10,000) - 1
        // We subtracted one since the base period takes one clock
        // cycle longer than the value we load in.
        dev.PwmBasePeriod=199;

        // Set up this PWM channel's duty period to take up 60% of
        // each base period cycle.  The base period takes 200 clock
        // cycles, so we want the duty period to take:
        //   200 * 0.60 = 120 clock cycles
        // Since the duty period takes one cycle longer than the value
        // we load into it, we specify 119 here:
        dev.PwmDutyPeriod[0]=119;


        // Put the PWM pin into output mode
        // PWM 0's output pin is on Port 2, bit 4
        dev.SetDirectionBit(2, 4, 1);

        // Enable the main PWM clock
        dev.PwmClockState=Eth32PwmClock.Enabled;

        // Finally, enable the PWM channel
        dev.PwmChannel[0]=Eth32PwmChannel.Normal;
```

```
}
catch (Eth32Exception e)
{
        // Handle Eth32 errors here
}
```

See Also

PwmBasePeriod Property, PwmChannel Property, SetPwmParameters Method

---

Readback Method

```
int Readback(int port)
```

Summary

This method retrieves (reads back) the current output value for the specified port. This is the value that was last written by calling the OutputByte Method or one or more calls to the OutputBit Method.

Parameters

- port - The port number to read back (0-5)

Return Value

This method returns an integer. The return value is the port's current output value.

See Also

ReadbackByte Method, OutputBit Method, OutputByte Method

ReadbackByte Method

```
byte ReadbackByte(int port)
```

Summary

This method is the same as the Readback Method but returning a byte. It is simply provided for convenience.

ResetDevice Method

```
ResetDevice()
```

Summary

This method resets most aspects of the device to their power-up default status. It does not perform a "cold reset" of the device. All TCP/IP connections to the device are preserved and do not need to be reestablished. See the remarks below for a list of everything that is affected.

Parameters

This method does not have any parameters.

Return Value

This method does not have a return value.

Remarks

The following parts of the device are reset by this method

- All digital I/O ports are configured as inputs.
- The output values of all digital I/O ports are set to zero.
- The LED’s are turned off
- The Analog to Digital Converter is disabled.
- The analog voltage reference is configured to the external reference (Eth32AnalogReference.External).
- The analog channel assignments are all set to the single-ended channels. Logical channel 0 is set to single-ended channel 0, Logical channel 1 to single-ended 1, and so on.
- All events are disabled for all connections.
- Analog event definitions are cleared.
- Both counters are disabled.
- Counter values are set to zero.
- Counter rollover points are set to their highest possible values (0xFFFF hex for 16-bit counter 0, 0xFF hex for 8-bit counter 1).
- Counter event threshold (applies only to counter 0) set to zero.
- PWM channels are disabled and the main PWM clock is stopped.
- The PWM base period is set to its highest (lowest frequency) setting of 0xFFFF counts.

- The duty period of both PWM channels is set to zero.
- The connection flags are reset only for the connection that performed the reset. The connection flags for any other connections are not affected.

SerialNum Property

```
string SerialNum
```

Summary

This read-only property retrieves the serial number of the ETH32 device in string format as it is printed on the device.

Parameters

This property does not have any parameters.

Value

This property is a string. Its value is the string representation of the device's serial number.

Remarks

The serial number is made up of several components and arranged as follows:

```
(productid)-(batch)(unit)
```

where:

- productid is a number identifying the product type/model. This number is returned by the ProductID Property.
- batch is the batch number formatted as two letters. 1 becomes AA, 2 becomes AB, etc.
- unit is the unit number, zero padded to 3 digits if necessary.

See Also

ProductID Property

SetAnalogEventDef Method

```
void SetAnalogEventDef(int bank, int channel, int lomark, int himark, Eth32AnalogEvtDef defaultval)
```

Summary

This method defines the event thresholds for a single logical analog channel in the specified analog event bank. The thresholds that are defined determine what analog readings will cause the event to fire. The thresholds allow the event logic on the ETH32 device to assign a current state (high or low) to the event. The event will be considered high if the analog reading is at or above the given hi-mark and will be considered low if at or below the given lo-mark. Whenever the state of the event changes (low to high or high to low), an event notification will be fired. When the analog reading is between the lo-mark and hi-mark, it will retain its previous value. This allows "hysteresis" to be built into the event so that a fluctuating signal will not cause an event to continuously fire. The thresholds are specified in 8-bit resolution, and thus they will be compared with the eight Most Significant Bits of the analog readings to determine when an event should be fired. The given hi-mark must be greater than the lo-mark.

Normally, the "initial state" (high or low) of the analog event definition is determined by the current level of the analog reading at the time the event definition is defined. However, if the current analog reading is between the lo-mark and hi-mark, an initial state cannot be accurately assigned. To deal with this, this method accepts a parameter that defines a default state to be used when the initial state cannot be determined. In all other situations (when the reading at the time of event definition is <= lo-mark or >= hi-mark) this parameter will simply be ignored.

Parameters

- bank - Specifies the event bank (0 or 1).
- channel - Specifies the logical channel (0-7).
- lomark - Low threshold, 8 Most Significant Bits (0-255).
- himark - High threshold, 8 Most Significant Bits (0-255).
- defaultval - If the current reading is between lomark and himark, this specifies whether the event should be considered high or low to begin with. Otherwise, this parameter is ignored. This parameter is a Eth32AnalogEvtDef enumerator type, which has the following valid values:
    - Eth32AnalogEvtDef.Low - Consider the channel to be low
    - Eth32AnalogEvtDef.High - Consider the channel to be high

Return Value

This method does not return a value.

Remarks

Please note that defining the thresholds with this method does not enable the current connection to actually receive the event notifications when they occur. These must be enabled using the EnableEvent Method. Also note that the analog event thresholds are common to all connections. Changing the thresholds will affect other connections if they are utilizing that particular event.

Because the ETH32 device has two analog event banks, two events can be defined for each logical analog channel on the board. Applications can utilize both event banks independently to implement a number of different event notification schemes.

Example

```
Eth32 dev = new Eth32();
int lomark;
int himark;

try
{
        // .... Your code that establishes a connection here

        // .... Your code that sets up your event handler
        // function goes here (or later)

        // Enable the Analog to Digital Converter
        dev.AnalogState=Eth32AnalogState.Enabled;

        // Configure logical channel 7 to read the physical channel 7 relative to ground (single-ended)
        // This is the power-on default anyway, but is shown here for completeness:
        dev.AnalogAssignment[7]=Eth32AnalogChannel.SE7;

        // Configure the analog voltage reference to be the internal 5V source
        dev.AnalogReference=Eth32AnalogReference.Internal;


        // Define an event that fires when channel 7 goes above 3.5V or
        // falls below 3.0V.  Remember that the thresholds must be calculated
        // knowing the voltage reference (in this case 5V).  They also must be
        // converted to the 8 Most Significant Bits from 10-bit by dividing by 4.
        // If the current reading happens to be between the low and high threshold,
        // we will default to the event starting out low.
        lomark=(int)(3.0 / 5.0 * 1024 / 4);
        himark=(int)(3.5 / 5.0 * 1024 / 4);
        dev.SetAnalogEventDef(0, 7, lomark, himark, Eth32AnalogEvtDef.Low);

        // Now that an event is defined in bank 0, channel 7, enable receiving
        // events from it.
        // We'll give this event an arbitrary ID of 8000
        dev.EnableEvent(Eth32EventType.Analog, 0, 7, 8000);

        // You will now receive events when channel 7 crosses the threshold
        // to being over 3.5V or crosses to under 3.0V.
}
catch (Eth32Exception e)
{
        // Handle Eth32 errors here
}
```

See Also

EnableEvent Method, GetAnalogEventDef Method, InputAnalog Method

SetDirection Method

```
void SetDirection(int port, int direction)
```

Summary

This method sets the direction register for a digital I/O port, which configures each pin (bit) of the port as an input or output. The direction of each bit of the port can be set individually, meaning that some bits of the port can be inputs at the same time that other bits on the same port are outputs. A 1-bit in the direction register causes the corresponding bit of the port to be put into output mode, while a 0-bit specifies input mode. For example, a value of 0xF0 would put bits 0-3 into input mode and bits 4-7 into output mode.

Parameters

- port - The port number (0-5).
- direction - The new direction register for the port.

Return Value

This method does not return a value.

Remarks

Port 3 shares its pins with the analog channels. When the ADC is enabled, all pins of port 3 are forced into input mode. The direction register of port 3 cannot be modified while the ADC is enabled.

The valid range for the direction parameter is any 8-bit number (ranges from 0 to 255). However, note that because ports 4 and 5 are single-bit ports, only bit 0 will have any effect on those ports.

For your convenience, constants for the direction parameter are provided that configure the port bits to be all inputs or all outputs. These are, respectively, Eth32.DirInput and Eth32.DirOutput.

Example

```
Eth32 dev = new Eth32();

try
{
        // .... Your code that establishes a connection here

        // Configure all odd bits of port 0 as inputs and even bits as outputs
        //  Direction parameter of 10101010 binary, which is 0xAA hex or 170 decimal
        dev.SetDirection(0, 170);
}
catch (Eth32Exception e)
{
        // Handle Eth32 errors here
}
```

See Also

GetDirection Method, GetDirectionBit Method, SetDirectionBit Method

---

SetDirectionBit Method

```
void SetDirectionBit(int port, int bit, int direction)
void SetDirectionBit(int port, int bit, bool direction)
```

Summary

This overloaded method alters a single bit of a port's direction register without affecting the value of any other bits. See the SetDirection Method for further description of the direction register.

Parameters

- port - The port number (0-5).
- bit - Which bit within the port to alter (0-7).
- direction - Make the bit an input (0 or false) or an output (1 or true).

Return Value

This method does not return a value.

Remarks

This method alters the specified direction register bit in a single operation directly on the ETH32 device. In other words, it does NOT read the current value over the network, modify it and then write it back. By doing it in a single operation, this avoids the potential of inadvertently overwriting changes made to other bits by other connections.

Port 3 shares its pins with the analog channels. When the ADC is enabled, all pins of port 3 are forced into input mode. The direction register of port 3 cannot be modified while the ADC is enabled.

See Also

GetDirection Method, GetDirectionBit Method, SetDirection Method

---

SetEeprom Method

```
void SetEeprom(int address, int length, byte[] buffer)
```

Summary

This method stores data into the non-volatile EEPROM memory of the device. Writing to EEPROM memory is a relatively slow process, which will temporarily disrupt event monitoring on the device. See the user manual for specific timing information.

Parameters

- address - The starting location to which data should be stored (0-255).
- length - The number of bytes to store. Valid values for this parameter depend on what is provided for the address parameter. For example, with an address of 0, you may specify a length of all 256 bytes, but with an address of 255, length may only be 1.
- buffer - The data to store into EEPROM memory. This must contain at least as many bytes as you are requesting to store.

Return Value

This method does not return a value.

See Also

GetEeprom Method

SetPwmParameters Method

```
void SetPwmParameters(int channel, Eth32PwmChannel state, double freq, double duty)
```

Summary

This method is provided for your convenience in working with all of the various PWM settings. It allows you to specify the PWM frequency and the duty cycle of a channel in more familiar terms (hertz and percentage) rather than PWM clock counts. It also puts the appropriate I/O pin into output mode unless you specify that the PWM channel should be disabled. This method internally calls several other API functions to set everything up, therefore replacing calls to PwmBasePeriod Property, PwmDutyPeriod Property, PwmClockState Property, PwmChannel Property, and SetDirectionBit Method with a single call to this method.

Parameters

- channel - Specifies the PWM channel number (0 or 1).
- state - This property is a Eth32PwmChannel enumerator type, which has the following valid values:
  - Eth32PwmChannel.Disabled - The PWM pin will function as a normal digital I/O pin.

  - Eth32PwmChannel.Normal - The PWM pin will function as a PWM output. It will be high for the time specified by the duty period and low for the rest of the PWM base period.
  - Eth32PwmChannel.Inverted - The PWM pin will function as a PWM output. It will be low for the time specified by the duty period and high for the rest of the PWM base period.

- freq - Specifies the frequency in Hertz. The valid range is 30.5 HZ to 40,000 HZ (40 KHZ)
- duty - Specifies the duty cycle as a percentage (A floating point number from 0.0 to 1.0). This specifies the percentage of each cycle that the channel will be active.

Return Value

This method does not return a value.

Remarks

Note that this method uses the PwmBasePeriod Property to set the PWM base period. Because the PWM base period is shared between both PWM channels, this will affect the other PWM channel if you specify a frequency different than what is already in effect.

Example

```
Eth32 dev = new Eth32();

try
{
        // .... Your code that establishes a connection here

        // Set up PWM channel 0 to have a 10 KHZ, 60% PWM signal:
        dev.SetPwmParameters(0, Eth32PwmChannel.Normal, 10000, 0.60);

}
catch (Eth32Exception e)
{
        // Handle Eth32 errors here
}
```

See Also

GetPwmParameters Method

Timeout Property

```
int Timeout
```

Summary

This property configures the internal API timeout on any operations that require a response from the ETH32 device (for example, InputByte). If a method or property routine does not receive a reply from the ETH32 within the timeout period specified, it raises an Eth32Exception with an ErrorCode of EthError.Timeout error. This property does not affect the actual ETH32 device, but just the functionality within the API itself. This property does not affect any other Eth32 objects that may be open.

Parameters

This property does not have any parameters.

Value

This property is an integer. It specifies the timeout in milliseconds. A value of zero means that operations should never time out.

VerifyConnection Method

```
void VerifyConnection()
```

Summary

This method sends a "ping" command (not an ICMP Ping) to the ETH32 device and waits for a response. It can be used to verify that there is still a good connection to the device.

Parameters

This method does not have any parameters.

Return Value

This method does not return a value. If any error occurs, an Eth32Exception will be raised.

See Also

Connect Method, Connected Property, Disconnect Method

Event Handler

When hardware events occur on the ETH32, information about that event is transmitted to your application if you have enabled it using the EnableEvent Method. When this information is received, the .NET Eth32 class notifies your application of the event in a manner that is consistent with the way .NET applications typically process events. That is, the ETH32 events will be processed by your application in a manner very similar to the way that the Click event of a form’s command button would be processed.

The event handler function is a function written by you, the programmer. Because it is a function you write, you have complete freedom to inspect whichever aspects of the event data you need to and react however you see fit. All of the information about the event is contained in the second parameter to the function, the EventArgs object. An Eth32EventArgs object (which is derived from EventArgs) is passed to this parameter. The Eth32EventArgs object has a member called event_info, which is a eth32_event structure containing all of the event information.

Your event handler function will be executed by a separate thread. You should be aware of this fact if you will be doing any tasks in your function that are not thread safe. The Eth32 class waits for your function to return before calling it again with the next event. Therefore, be aware that if you perform any long operations, it will delay more events from being processed. Note that each Eth32 object has its own event thread, so if you are using a single event handler function for multiple connections (objects), be aware that at times there may be more than one instance of your function executing.

Writing and Registering an Event Handler

Your event handler function may be given any name, but regardless of its name, it must accept specific parameter types. The syntax of the event handler function and the code used to register it with an Eth32 object are shown below. These are not complete examples, in that the code fragments shown here would typically be within one of your classes in your application. These code fragments show functions declared as Private and Static (Shared in Visual Basic), but neither of those options are required if other options work better in your application.

- Visual C#.NET

```
private static void MyEth32EventHandler(Object s, EventArgs e)
{
        // If this function was called by Eth32 event processing, then
        // s (sender) is actually the Eth32 object that received the event
        // and e is actually an Eth32EventArgs object.  But to be safe,
        // we'll make sure by using this code:
        Eth32 sender;
        Eth32EventArgs args;

        sender = s as Eth32;
        if(sender == null)
        {
                // s wasn't really an Eth32 object - quit
                return;
        }

        args = e as Eth32EventArgs;
        if(args == null)
        {
                // The arguments weren't really Eth32EventArgs - quit
                return;
        }

        // You may now easily access the Eth32 object by using sender and
        // the event information by using args.event_info
}
```

```
// Later in your code, when you want to register your event handler,
// assuming dev is your Eth32 object:
dev.HardwareEvent += new EventHandler(MyEth32EventHandler);
```

- Visual Basic.NET

```
Private Shared Sub MyEth32EventHandler(ByVal s As System.Object, ByVal e As System.EventArgs)
        ' If this function was called by Eth32 event processing, then
        ' s (sender) is actually the Eth32 object that received the event
        ' and e is actually an Eth32EventArgs object.  But to be safe,
        ' we'll make sure by using this code:

        Dim args As Eth32EventArgs
        Dim sender As Eth32


        If TypeOf s Is Eth32 Then
                sender = CType(s, Eth32)
        Else
                ' s wasn't really an Eth32 object - quit
                Exit Sub
        End If

        If TypeOf e Is Eth32EventArgs Then
                args = CType(e, Eth32EventArgs)
        Else
                ' The arguments weren't really Eth32EventArgs - quit
                Exit Sub
        End If

        ' You may now easily access the Eth32 object by using sender and
        ' the event information by using args.event_info

End Sub

' Later in your code, when you want to register your event handler,
' assuming dev is your Eth32 object:
AddHandler dev.HardwareEvent, AddressOf MyEth32EventHandler
```

- Visual C++.NET (Managed)

```
private:
static void MyEth32EventHandler(Object* s, EventArgs* e)
{
        // If this function was called by Eth32 event processing, then
        // s (sender) is actually the Eth32 object that received the event
        // and e is actually an Eth32EventArgs object.  But to be safe,
        // we'll make sure by using this code:
        Eth32 *sender;
        Eth32EventArgs *args;

        sender = dynamic_cast<Eth32*>(s);
        if(sender == NULL)
        {
                // s wasn't really an Eth32 object - quit
                return;
        }

        args = dynamic_cast<Eth32EventArgs*>(e);
```

```
            if(args == NULL)
            {
                    // The arguments weren't really Eth32EventArgs - quit
                    return;
            }

            // You may now easily access the Eth32 object by using sender and
            // the event information by using args->event_info
    }

    // Later in your code, when you want to register your event handler,
    // assuming dev is your Eth32 object pointer:
    dev->HardwareEvent += new EventHandler(0, &MyEth32EventHandler);
```

Example

This very simple, yet compilable C# example demonstrates the basics of writing and registering an event handler function with the ETH32.

```
using System;
using WinfordEthIO;

public class MyExample
{
        private static void MyEth32EventHandler(Object s, EventArgs e)
        {
                // If this function was called by Eth32 event processing, then
                // s (sender) is actually the Eth32 object that received the event
                // and e is actually an Eth32EventArgs object.  But to be safe,
                // we'll make sure by using this code:
                Eth32 sender;
                Eth32EventArgs args;

                sender = s as Eth32;
                if(sender == null)
                {
                        // s wasn't really an Eth32 object - quit
                        return;
                }

                args = e as Eth32EventArgs;
                if(args == null)
                {
                        // The arguments weren't really Eth32EventArgs - quit
                        return;
                }


                switch(args.event_info.id)
                {
                        case 1000:
                                // React accordingly to Port 1, Bit 3 event
                                // We'll turn on the ETH32's LED 0 if the bit value is high and
                                // turn it off otherwise.
                                if(args.event_info.val != 0)
                                        sender.Led[0]=true;
                                else
                                        sender.Led[0]=false;
                                break;
                        case 1001:
                                // React accordingly to Port 1, Bit 4 event
                                // We'll turn on the ETH32's LED 1 if the bit value is high and
                                // turn it off otherwise.
```

```
                        if(args.event_info.val != 0)
                                sender.Led[1]=true;
                        else
                                sender.Led[1]=false;
                        break;
                }
        }


        static void Main()
        {
                try
                {
                        Eth32 dev;

                        dev = new Eth32();

                        // NOTE: Substitute the IP address or DNS name of your device here.
                        dev.Connect("192.168.1.100");

                        // Register our event handler function to handle all incoming events
                        dev.HardwareEvent += new EventHandler(MyEth32EventHandler);

                        // Enable events on Port 1, bits 3 and 4
                        // But first, if these are pushbuttons and need a pullup resistor
                        // to make them float high, enable that:
                        dev.OutputBit(1, 3, 1);
                        dev.OutputBit(1, 4, 1);
                        // Now enable the event:
                        dev.EnableEvent(Eth32EventType.Digital, 1, 3, 1000);
                        dev.EnableEvent(Eth32EventType.Digital, 1, 4, 1001);


                        // Display a MessageBox.  This function will not return until the
                        // user clicks OK, so it will keep the application running until then.
                        System.Windows.Forms.MessageBox.Show("When you're finished monitoring events, "
                                + "click OK to end application.");
                }
                catch (Eth32Exception e)
                {
                        // Handle Eth32 errors here
                        System.Windows.Forms.MessageBox.Show("Eth32 exception: " + e.ToString());
                }
        }
}
```

Configuration / Detection Functionality

Most of the network configuration and detection functionality of the ETH32 API is contained in the Eth32Config class. If plugins are used to find information about the PC's network interfaces and/or to utilize a sniffer library, that functionality is provided by the Eth32ConfigPlugin class. These classes, their members, and associated structures are described below.

Error Handling

Error codes for the Configuration / Detection Functionality are defined in the EthError enumerator along with the error codes for the main API. Error codes can be translated into a string using the static ErrorString Method of the main Eth32 class.

Structures

eth32cfg_ip Structure

The eth32cfg_ip structure holds an IP address in binary form. It is used to represent IP address information in the ETH32 device configuration structure, to specify the broadcast address, and to retrieve IP address information about the PC's network interfaces.

```
public struct eth32cfg_ip
{
        public byte[] bytes;
        public eth32cfg_ip(bool initarray){}
        public override string ToString(){}
}
```

- bytes - Array containing individual octets of the IP address. If you manually create this array, it must have a length of four elements. Index 0 contains the most significant, e.g. 192 from the address 192.168.1.100
- Constructor(bool) - If true is passed to this constructor, an array of four elements is created and initialized to zeroes.
- ToString - Returns a string representation of the IP address.

eth32cfg_mac Structure

The eth32cfg_mac structure holds a MAC address in binary form. It is used within the ETH32 device configuration structure.

```
public struct eth32cfg_mac
{
        public byte[] bytes;
        public eth32cfg_mac(bool initarray){}
        public override string ToString(){}
}
```

- bytes - Array containing individual octets of the MAC address. If you manually create this array, it must have a length of six elements. Index 0 contains the first and most significant octet.
- Constructor(bool) - If true is passed to this constructor, an array of six elements is created and initialized to zeroes.
- ToString - Returns a string representation of the MAC address.

eth32cfg_data Structure

The eth32cfg_data structure holds all of the network configuration and device information data for a particular ETH32 device. It is provided to your application when retrieving information about detected devices. Your application can also fill in or modify the information and provide it to the API to store new configuration into a device.

```
public struct eth32cfg_data
{
        public byte product_id;
        public byte firmware_major;
        public byte firmware_minor;
        public byte config_enable;
        public eth32cfg_mac mac;
        public ushort serialnum_batch;
        public ushort serialnum_unit;
        public eth32cfg_ip config_ip;
        public eth32cfg_ip config_gateway;
        public eth32cfg_ip config_netmask;
        public eth32cfg_ip active_ip;
        public eth32cfg_ip active_gateway;
        public eth32cfg_ip active_netmask;
        public byte dhcp;
}
```

- product_id - Contains the product ID code for the device. This will be 105 for ETH32 devices. This makes up a portion of the serial number.
- firmware_major - Contains the major portion of the firmware version, e.g. 3 from 3.000
- firmware_minor - Contains the minor portion of the firmware version, e.g. 0 from 3.000
- config_enable - Nonzero if the device's Allow Config switch is set to Yes
- mac - The MAC address of the device
- serialnum_batch - The batch number portion of the device's serial number
- serialnum_unit - The unit number portion of the device's serial number
- config_ip - The static IP address stored in the device. This is ignored if DHCP is active.
- config_gateway - The static gateway IP address stored in the device. This is ignored if DHCP is active.
- config_netmask - The static network mask stored in the device. This is ignored if DHCP is active.
- active_ip - The IP address being used by the device, whether it was provided by DHCP or statically configured.
- active_gateway - The gateway IP address being used by the device, whether it was provided by DHCP or statically configured.
- active_netmask - The network mask being used by the device, whether it was provided by DHCP or statically configured.
- dhcp - Nonzero if DHCP is being used by the device, or zero if the static settings (config_ip, etc) are being used.

If a device is using DHCP, then active_ip will most likely be different than the static (stored) config_ip, and so on for the gateway and netmask. If DHCP is not being used, then active_ip will be the same as config_ip, and so on for the gateway and netmask.

When using this structure with the SetConfig Method, you may modify the config_ip, config_gateway, config_netmask, and dhcp members in order to update the corresponding settings within the ETH32 device. The other members of the structure should not be modified, since they will either be ignored, or are required for the new configuration to be accepted by the device. Specifically, the MAC address and serial number information must match the device's information, or the device will ignore the new configuration data.

Eth32ConfigPluginInterface Structure

The Eth32ConfigPluginInterface structure holds information about a network interface card of the PC. This information can be provided by a plugin loaded into the ETH32 API.

```
public struct Eth32ConfigPluginInterface
{
    public eth32cfg_ip Ip;
    public eth32cfg_ip Netmask;
    public Eth32ConfigInterfaceType InterfaceType;
    public string StandardName;
    public string FriendlyName;
    public string Description;
}
```

- Ip - The IP address of the network interface
- Netmask - The network mask of the network interface
- InterfaceType - The type of network that this network interface is for. This can be one of these values:
    - Eth32ConfigInterfaceType.Unknown - This is used if the current plugin doesn't provide information about the network interface type.
    - Eth32ConfigInterfaceType.Other - This is used if the plugin provides information about the interface type, but it isn't one of the predefined constants.
    - Eth32ConfigInterfaceType.Ethernet - Ethernet interface
    - Eth32ConfigInterfaceType.Tokenring - Token Ring interface
    - Eth32ConfigInterfaceType.Fddi - FDDI (Fiber Distributed Data Interface) interface
    - Eth32ConfigInterfaceType.Ppp - PPP (Point-to-Point Protocol) interface
    - Eth32ConfigInterfaceType.Loopback - Local loopback interface (e.g. 127.0.0.1)

  - Eth32ConfigInterfaceType.Slip - SLIP (Serial Line Internet Protocol) interface

- StandardName - This is typically an internal identifier string that identifies the interface, but is not very human-readable. The exact value depends on the plugin being used.
- FriendlyName - The human-readable name for the interface. For example, Local Area Connection. This member will be empty when the WinPcap plugin is being used.
- Description - A description of the interface. The value of this member depends on the plugin being used, but typically includes the manufacturer or model of the card. This member will be available when using the System plugin or when using the WinPcap plugin.

Eth32Config Member Reference

BroadcastAddress Property

```
eth32cfg_ip BroadcastAddress
```

Summary

This read/write property defines the broadcast address that will be used when sending out queries or new configuration data to ETH32 devices. It defaults to 255.255.255.255, which works well in most situations.

Parameters

This property does not have any parameters.

Value

This property is a eth32cfg_ip Structure.

See Also

BroadcastAddressString Property

BroadcastAddressString Property

```
string BroadcastAddressString
```

Summary

This read/write property returns or alters the same information as the BroadcastAddress Property, but in string format.

Parameters

This property does not have any parameters.

Value

This property is a string representation of the broadcast address.

See Also

BroadcastAddress Property

DiscoverIp Method

```
int DiscoverIp(eth32cfg_mac mac)
int DiscoverIp(byte product_id, ushort serialnum_batch, ushort serialnum_unit)
int DiscoverIp(eth32cfg_mac mac, byte product_id, ushort serialnum_batch, ushort serialnum_unit)
int DiscoverIp(Eth32ConfigFilter filter, eth32cfg_mac mac, byte product_id,
               ushort serialnum_batch, ushort serialnum_unit)
```

Summary

This overloaded method is used to detect ETH32 devices and their currently-active IP configuration settings. This method allows you to specify a filter so that only the information for the specific ETH32 device that you are interested in will be returned (in case there are multiple ETH32s on the network). This is intended for applications that need to discover the IP of a device that is using DHCP to get its IP address. This method uses a new command to the ETH32 device that is only supported by devices with firmware v3.000 and on. Any older devices on the network will not be detected. The eth32cfg_data structure for devices detected with this method will not have all fields filled in, since the response from the ETH32 does not include all available information. Only the product_id, mac, serialnum_batch, serialnum_unit, active_ip, active_gateway, active_netmask, and dhcp fields will be filled in and valid.

There are four different overloaded variants of this method. If only a MAC address is provided, devices will be discovered based only on MAC address. If the product_id, serialnum_batch, and serialnum_unit parameters are provided, the device will be discovered based only on the serial number (those three items make up the serial number). If MAC and serial number information is provided, only a device that matches both will be discovered. Finally, the last variant includes a filter parameter, that instructs the method which data to filter on. Although this variant includes parameters for both MAC and serial number information, it will only be considered if the appropriate flag is present in the filter parameter.

Once this method returns, the configuration data for any devices that have been found will be available through the Result Property.

Parameters

- filter - Specifies which parameters should be considered in discovering the device. If more than one flag is specified, then the device must match BOTH. This parameter is a Eth32ConfigFilter enumerator type, which has the following valid values:
    - Eth32ConfigFilter.None - The parameters will be ignored. All devices will be discovered.
    - Eth32ConfigFilter.Mac - Only devices matching the provided MAC address will be discovered.

  - Eth32ConfigFilter.Serial - Only devices matching the provided serial number information (id, batch, unit) will be discovered.

- mac - The MAC address of the device you are trying to discover
- product_id - The product ID code (part of the serial number) of the device you are trying to discover. For ETH32 devices, this is 105.
- serialnum_batch - The batch number portion of the serial number for the device you are trying to discover.
- serialnum_unit - The unit number portion of the serial number for the device you are trying to discover.

Return Value

This method returns the number devices that have been found.

Remarks

The number of devices that were found is returned by the method, but also remains available from the NumResults Property. When you are finished with the results, you may free the memory associated with the results using the Free Method. This is done automatically for you if the object is destroyed, or if you call the DiscoverIp Method or the Query Method again on the same object. Note that this means each Eth32Config object holds only one active set of results at one time.

Example

```
Eth32Config devdetect = new Eth32Config();

try
{
    // Set broadcast address - this line would not
    // be necessary since 255.255.255.255 is the default anyway
    devdetect.BroadcastAddressString = "255.255.255.255";

    // Find a device by serial number -- we can use the ProductId constant
    // 1 for the batch (AB), and 82 for the unit number.
    // This would be serial number 105-AB082 as shown on the device.
    devdetect.DiscoverIp(Eth32Config.ProductId, 1, 82);

    if (devdetect.NumResults == 0)
        MessageBox.Show("Device not found");
    else
    {
        // Device was found -- here's a quick example of using the information to now
        // connect to the device and turn on LED 0.
        Eth32 dev = new Eth32();
        dev.Connect(devdetect.Result[0].active_ip.ToString());
        dev.Led[0] = true;
    }
}
```

```
catch (Eth32Exception err)
{
    // Handle errors here
}
```

See Also

Result Property, NumResults Property, Query Method, Free Method

Free Method

```
void Free()
```

Summary

This method frees any memory associated with the current set of results held by the object. This can be called after you are finished with the results from the DiscoverIp Method or the Query Method. However, it is called automatically for you when either of those methods is called again, as well as at the time the object is destroyed.

Parameters

This method does not have any parameters.

Return Value

This method does not return a value.

See Also

DiscoverIp Method, Query Method

IpConvert Method

```
static eth32cfg_ip IpConvert(string ipaddr)
static eth32cfg_ip IpConvert(System.Net.IPAddress ipaddr)
```

Summary

This overloaded method converts either a string representation of an IP address, or .Net's IPAddress object into the eth32cfg_ip binary representation of an IP address. If a string representation doesn't contain a valid IP address, or if the IPAddress object is the wrong type of address, an EthError.InvalidIp error will be raised.

Parameters

- ipaddr - The IP address to be converted.

Return Value

This method returns an eth32cfg_ip structure with the converted IP address.

See Also

IpConvertToNetIPAddress Method, IpConvertToString Method

IpConvertToNetIPAddress Method

```
static System.Net.IPAddress IpConvertToNetIPAddress(eth32cfg_ip ipbinary)
```

Summary

This method converts the eth32cfg_ip binary representation of an IP address into .Net's IPAddress object.

Parameters

- ipbinary - The IP address to be converted.

Return Value

This method returns an IPAddress object with the converted IP address.

See Also

IpConvert Method, IpConvertToString Method

IpConvertToString Method

```
static string IpConvertToString(eth32cfg_ip ipbinary)
```

Summary

This method converts the eth32cfg_ip binary representation into a string.

Parameters

- ipbinary - The IP address to be converted.

Return Value

This method returns a string representation of the converted IP address.

See Also

IpConvert Method, IpConvertToNetIPAddress Method

MacConvert Method

```
static eth32cfg_mac MacConvert(string macstring)
```

Summary

This method converts a string representation of a MAC address into the eth32cfg_mac binary representation of a MAC address. If the string doesn't contain a valid MAC address, an EthError.InvalidOther error will be raised.

Parameters

- macstring - The MAC address string to be converted.

Return Value

This method returns an eth32cfg_mac structure with the converted MAC address.

See Also

MacConvertToString Method

MacConvertToString Method

```
static string MacConvertToString(eth32cfg_mac macbinary)
```

Summary

This method converts an eth32cfg_mac binary representation of a MAC address into a string.

Parameters

- macbinary - The MAC address to be converted.

Return Value

This method returns a string representation of the MAC address.

See Also

MacConvert Method

NumResults Property

```
int NumResults
```

Summary

This read-only property indicates how many ETH32 devices were found the last time the DiscoverIp Method or the Query Method was called.

Parameters

This property does not have any parameters.

Value

This property is an integer. The value indicates the number of devices found, and therefore how many items are available through the Result Property.

See Also

Result Property

ProductId Constant

```
const byte ProductId = 105
```

Value

The ProductId constant defines the product ID code for ETH32 devices. This is one component of each device's serial number.

Query Method

```
int Query()
```

Summary

This method is used to detect all ETH32 devices on the local network segment and all of their available device information and configuration settings. Once this method returns, the configuration data for any devices that have been found will be available through the Result Property.

Parameters

This method does not have any parameters.

Return Value

This method returns the number devices that have been found.

Remarks

The number of devices that were found is returned by the method, but also remains available from the NumResults Property. When you are finished with the results, you may free the memory associated with the results using the Free Method. This is done automatically for you if the object is destroyed, or if you call the DiscoverIp Method or the Query Method again on the same object. Note that this means each Eth32Config object holds only one active set of results at one time.

As opposed to the DiscoverIp Method, which is only supported by devices with firmware 3.000 and greater, the Query Method detects all devices with all firmware versions. This method sends several queries out repeatedly in case any queries or responses are lost on the network. It also delays for a short while to listen for responses. Because of this, the DiscoverIp Method method will be faster if you are looking for a specific device, know its MAC address or serial number, and know it is running firmware v3.000 or greater.

Example

```
Eth32Config devdetect = new Eth32Config();
int i;

try
{
    // Set broadcast address - this line would not
    // be necessary since 255.255.255.255 is the default anyway
    devdetect.BroadcastAddressString = "255.255.255.255";

    // Find all devices
    devdetect.Query();

    if (devdetect.NumResults == 0)
        MessageBox.Show("No devices were found.");
    else
    {
        for (i = 0; i < devdetect.NumResults; i++)
        {
            MessageBox.Show("Device found with IP address of: " +
                            devdetect.Result[i].active_ip.ToString());
        }
    }
}
catch (Eth32Exception err)
{
    // Handle errors here
}
```

See Also

Result Property, NumResults Property, DiscoverIp Method, Free Method

Result Property

```
eth32cfg_data Result[int index]
```

Summary

This property is used to access the device information and configuration data for each device that was found on the last call to the Query Method or the DiscoverIp Method.

Parameters

- index - The index of the result to return.

Value

This property is a eth32cfg_data Structure. It returns the configuration data for the result at the specified index location.

Remarks

The index is zero-based, which means it can range from zero up to one less than the number of available results (as indicated by the NumResults Property).

See Also

eth32cfg_data Structure, NumResults Property, DiscoverIp Method, Query Method

SerialNumString Method

```
static string SerialNumString(byte product_id, ushort serialnum_batch, ushort serialnum_unit)
```

Summary

This method takes the numeric components of the ETH32 serial number and formats a serial number string in the same way that it is printed on the ETH32 device enclosure.

Parameters

- product_id - The product ID portion of the serial number
- serialnum_batch - The batch number portion of the serial number
- serialnum_unit - The unit number portion of the serial number

Return Value

This method returns a string representation of the serial number.

See Also

eth32cfg_data Structure

SetConfig Method

```
void SetConfig(eth32cfg_data config_data)
```

Summary

This method is used to store new configuration settings into an ETH32 device. The device's Allow Config switch must be set to Yes, or the new configuration will be rejected.

Parameters

- config_data - The new configuration data and product identification information

Return Value

This method does not return a value. If any error occurs, an Eth32Exception will be raised.

Remarks

The MAC address and serial number information members of the eth32cfg_data Structure identify which device is to be configured. If those members are not set correctly, the device will simply ignore the settings, or worst-case, if they match a different device you were not intending to configure, that device will accept the new configuration. Therefore, in most cases, although it is not required, it is best to take the eth32cfg_data Structure from the Result Property, modify as needed, and then provide that to this method.

Under normal circumstances, the device will accept the configuration and return a confirmation packet, which will cause the method to immediately return without raising any errors. If the device's Allow Config switch is set to No, it will return a rejection packet, which will cause the method to raise the EthError.ConfigReject error. If no response is received from the device, the method will raise the EthError.ConfigNoAck error after a short timeout.

See Also

eth32cfg_data Structure

Eth32ConfigPlugin Member Reference

ChooseInterface Method

```
void ChooseInterface(int index)
```

Summary

This method selects one of the available network interfaces on the PC as the interface on which the ETH32 Configuration / Detection API (Eth32Config class) should sniff for responses from ETH32 devices. This does not affect the main API functionality (the Eth32 class). The interface list must have been previously obtained using the GetInterfaces Method and the provided index must be a valid index within that list. Currently, this function is only applicable when the WinPcap plugin is loaded. Otherwise, the EthError.NotSupported will be raised.

Parameters

- index - The index of the interface in the previously-obtained interface list which should be chosen for sniffing responses

Return Value

This method does not return a value.

See Also

GetInterfaces Method

Free Method

```
void Free()
```

Summary

This method frees any memory associated with a the network interface list previously obtained using the GetInterfaces Method. This is done automatically if the GetInterfaces Method is called again later, but note that you must call Free on any Eth32ConfigPlugin objects in the same application process (if they have called the GetInterfaces Method) before loading a different plugin with the Load Method.

Parameters

This method does not have any parameters.

Return Value

This method does not return a value.

See Also

GetInterfaces Method

GetInterfaces Method

```
int GetInterfaces()
```

Summary

This method loads the list of available network interface cards on the PC. A plugin which provides this functionality must be loaded first before calling this method. This functionality is provided by both the System and the WinPcap plugins, but not by the None plugin. Once the method returns, details of each interface can be accessed through the NetworkInterface Property

Parameters

This method does not have any parameters.

Return Value

This method returns the number of network interface cards in the list. This number will also remain available from the NumInterfaces Property.

Remarks

If the currently-loaded plugin does not provide this functionality, an EthError.NotSupported error will be raised.

The memory used by the interface list can be freed with the Free Method. The only time this needs to be done manually is when one plugin (other than None) has been loaded, Eth32ConfigPlugin object(s) with interface list(s) are open, and you are getting ready to load a different plugin with the Load Method. This is due to the fact that the loaded plugin affects the entire process (note that the Load Method is static), so it is up to you as the programmer to ensure that any active Eth32ConfigPlugin objects are Free'd before changing the plugin.

See Also

Load Method, NetworkInterface Property, Free Method

Load Method

```
static void Load(Eth32ConfigPluginType plugin_type)
```

Summary

This method loads one of the pre-defined plugins. The currently-loaded plugin affects the entire process in terms of the Configuration and Detection functionality (the Eth32Config class), but does not affect the main functionality of the API (the Eth32 class). See the Plugins topic for more information.

Parameters

- plugin_type - The plugin to be loaded. This can be one of the following options:
  - Eth32ConfigPluginType.None - No plugin loaded. This is the default if Load is never called. If another plugin is loaded, calling Load with this option will remove the loaded plugin.
  - Eth32ConfigPluginType.System - The Windows API is used to provide information about the network interfaces on the PC. Using this plugin does not affect how queries are sent out or how responses are received.
  - Eth32ConfigPluginType.Pcap - The WinPcap library is used to provide information about the network interfaces as well as to sniff for ETH32 responses on the chosen interface.

Return Value

This method does not return a value.

Remarks

If a plugin is attempted to be loaded that is not present on the system, an EthError.NotSupported error will be raised.

When one plugin (other than None) has been loaded and Eth32ConfigPlugin object(s) with interface list(s) are open, you must make sure that the Free Method of each Eth32ConfigPlugin object is called before changing the plugin with this method. This is due to the fact that the loaded plugin affects the entire process (note that this method is static), so it is up to you as the programmer to ensure that any active Eth32ConfigPlugin objects are Free'd before changing the plugin.

See Also

Free Method

NetworkInterface Property

```
Eth32ConfigPluginInterface NetworkInterface[int index]
```

Summary

This read-only property provides access to the information about each of the network interfaces in the list, which must be previously obtained by calling the GetInterfaces Method.

Parameters

- index - The index of the interface within the list

Value

This property is a Eth32ConfigPluginInterface Structure. It returns the interface information for the result at the specified index location.

Remarks

The index is zero-based, which means it can range from zero up to one less than the number of available interfaces (as indicated by the NumInterfaces Property).

See Also

GetInterfaces Method, NumInterfaces Property

NumInterfaces Property

```
int NumInterfaces
```

Summary

This read-only property indicates how many network interfaces are in the list that was obtained by calling the GetInterfaces Method and which are now available through the NetworkInterface Property.

Parameters

This property does not have any parameters.

Value

This property is an integer. The value indicates the number of interfaces in the list.

See Also

GetInterfaces Method, NetworkInterface Property

Visual Basic 6

The Visual Basic 6 class provided by the ETH32 distribution is a "wrapper" class in that it depends on the core ETH32 API (eth32api.dll) for almost every action. While it internally uses the core API, it provides a more convenient way to use the API within Visual Basic 6. Particularly in the area of event processing, this class takes care of some of the behind-the-scenes details to make event handling easy to implement and consistent with the facilities provided by the language.

Getting Started

The Visual Basic 6 class is distributed with the ETH32 API as a set of three source files that you must add to your application. The following steps are necessary:

- Copy the files from the ETH32 distribution into your project directory (same directory as your source files). By default, the installer places these files in the C:\Program Files\winford\eth32\api\windows\vb6 directory. These files are also on the CD in the api\windows\vb6 directory. It includes five files:

  ```
  eth32.bas
  Eth32.cls
  eth32_form.frm
  Eth32Config.cls
  Eth32ConfigPlugin.cls
  ```

- Add the files to your project. This is done one file at a time using the "Add File..." option of the "Project" menu. To include all functionality, be sure to add all five files. If your application does not need the ability to detect or re-configure the network settings of ETH32 devices on the local network, you may omit the Eth32Config.cls and Eth32ConfigPlugin.cls files.

Basic Declaration

The main class that is provided is named Eth32. Because it provides an event that is fired when an event on the device fires, you must declare your object variable using the WithEvents keyword. Declaring an object WithEvents is not allowed in modules (.bas files) or in procedure-level variables, which means you should declare the object at the top of your form source code before any procedures, as follows (substituting any valid variable name for dev):

```
Dim WithEvents dev As Eth32
```

Later in your code, at the point you wish to instantiate the object and connect to the device, your code should be similar to the following (substituting the variable name you used in the Dim statement for dev and the actual address or DNS name of your device for 192.168.1.100)

```
Set dev = New Eth32
dev.Connect "192.168.1.100"
```

Error Handling

Errors that may occur within the class or core API cause errors to be raised in your application. This means that as you use the device, you do not need to check return values for error codes. Instead, if an error occurs, an error will be raised and the applicable error handling code you have designated (if any) will be executed. As a rule, you should include error handling code for your application so that, for example, if an attempt to connect to the device fails, it does not cause an unhandled exception (which causes the application to exit).

You should use the "On Error GoTo" statements to install error handlers as is done for any other VB 5/6 error handling. When an error occurs and your error handling code executes, if the error was raised by the Eth32 class, the Err.Number variable will contain one of the possible error codes defined by the EthError enumerator. The following example illustrates the basic idea.

```
Dim WithEvents dev As Eth32

Private Function MyConnect() As Boolean
        ' If we connect successfully, return True.
        ' Otherwise, display an error message box and return False otherwise
    On Error GoTo err_handler

    Set dev = New Eth32

    dev.Connect "192.168.1.100"

    MyConnect=True
    Exit Function
err_handler:
    MsgBox "Error connecting to the ETH32 device: " & dev.ErrorString(Err.Number)
    MyConnect=False

End Function
```

Error Codes

Error code constants are defined by the EthError enumerator. The following error codes are defined:

- EthErrorNone - Success, no error.
- EthErrorGeneral - A miscellaneous or uncategorized error has occurred.
- EthErrorClosing - Function aborted because the device is being closed.
- EthErrorNetwork - Network communications error. Connection was unable to be established or existing connection was broken.
- EthErrorThread - Internal error occurred in the threads and synchronization library.
- EthErrorNotSupported - Function not supported by this device.
- EthErrorPipe - Internal API error dealing with data pipes.
- EthErrorRthread - Internal API error dealing with the "Reader thread."
- EthErrorEthread - Internal API error dealing with the "Event thread."
- EthErrorMalloc - Error dynamically allocating memory.
- EthErrorWindows - Internal API error specific to the Microsoft Windows platform.
- EthErrorWinsock - Internal API error in dealing with the Microsoft Winsock library.
- EthErrorNetworkIntr - Network read/write operation was interrupted.

- EthErrorWrongMode - Something is not configured correctly in order to allow this functionality.
- EthErrorBcastOpt - Error setting the SO_BROADCAST option on a socket.
- EthErrorReuseOpt - Error setting the SO_REUSEADDR option on a socket.
- EthErrorConfigNoack - Warning - device did not acknowledge our attempt to store a new configuration.
- EthErrorConfigReject - Device has rejected the new configuration data we attempted to store. Configuration switch on device may be disabled.
- EthErrorLoadlib - Error loading an external DLL library.
- EthErrorPlugin - General error with the currently configured plugin/sniffer library.
- EthErrorBufsize - A buffer provided was either invalid size or too small.
- EthErrorInvalidHandle - Invalid device handle was given.
- EthErrorInvalidPort - The given port number does not exist on this device.
- EthErrorInvalidBit - The given bit number does not exist on this port.
- EthErrorInvalidChannel - The given channel number does not exist on this device.
- EthErrorInvalidPointer - The pointer passed in to an API function was invalid.
- EthErrorInvalidOther - One of the parameters passed in to an API function was invalid.
- EthErrorInvalidValue - The given value is out of range for this I/O port, counter, etc.
- EthErrorInvalidIp - The IP address provided was invalid.
- EthErrorInvalidNetmask - The subnet mask provided was invalid.
- EthErrorInvalidIndex - Invalid index value.
- EthErrorTimeout - Operation timed out before it could be completed.
- EthErrorAlreadyConnected - An object that is already connected cannot have Connect called again.
- EthErrorNotConnected - This operation requires the object to be connected.

Structures (User Defined Types)

eth32_event

The eth32_event data type holds all of the information about an event that has fired. It is included in the arguments to your event handler function (see the Event Handler section).

```
Public Type eth32_event
  id As Long
  type As Long
  port As Long
  bit As Long
  prev_value As Long
  value As Long
  direction As Long
End Type
```

- id - The user-assigned event ID that you gave this event when enabling it.
- type - Event type, as defined by the Eth32EventType enumerator constants EVENT_DIGITAL, EVENT_ANALOG, EVENT_COUNTER_ROLLOVER, EVENT_COUNTER_THRESHOLD, and EVENT_HEARTBEAT
- port - For digital events, this specifies the port number the event occurred on. For analog events, it specifies the event bank number (0 or 1), and for counter events, it specifies which counter the event occurred on.
- bit - For a digital bit event, this specifies the bit number that changed. For an analog event, it specifies the analog channel, and for a digital port event, this will be -1.
- prev_value - The old value of the bit, port, or analog channel (as appropriate) before the event fired.
- value - The new value of the bit, port, or analog channel that caused the event to fire. In the case of counter events, this indicates the number of times the event occurred since the last time this event was fired (almost always 1).
- direction - Indicates whether the new value of the bit, port, or channel is greater or less than the previous value. It is 1 for greater than or -1 for less than.

Eth32 Member Reference

The members of the Eth32 class are described below. Several features of the ETH32 device are represented in the class as properties. When the property is read by your code, a request is sent to the ETH32 device, the ETH32 replies with the requested value, and that value is returned as the value of the property. When your code writes a new value to the property, a command is sent to the ETH32 storing the new value for that setting. All of the Eth32 class properties are both readable and writable unless otherwise specified.

Most of the example code below shows only the relevant code, not a complete compilable application. For the purposes of these examples, assume that the Eth32 object has a variable name of *dev* and is declared at the top of a form module as follows:

```
Dim WithEvents dev As Eth32
```

AnalogAssignment Property

```
Public Property AnalogAssignment(ByVal channel As Long) As Eth32AnalogChannel
```

Summary

When this property is written, it assigns a logical analog channel to one of the physical channels. When it is read, it returns the current physical channel assignment for the specified logical channel. The logical channel assignment specifies which physical pins are used to determine the value of the analog reading when that logical channel is read or monitored for events. There are eight logical channels, each of which may be arbitrarily assigned to physical channels using this property.

Parameters

- channel - The logical channel number (0-7).

Value

This property is a Eth32AnalogChannel value, which is an enumerator containing constants defining the possible channel assignments. The possible values of this enumerator are defined in the Remarks section below.

Remarks

The logical channels simply provide a way to select which of the many physical channel sources listed below will be continually updated for reading on the device and, if configured to do so, monitored for analog events.

The assignments given to the logical channels may be completely arbitrary. Also, it is permissible to have more than one logical analog channel assigned to the same physical channel source. This can occasionally be advantageous for event monitoring. Since there are two possible event definitions per logical channel, assigning more than one logical channel to the same physical channel allows more than two event definitions on that physical channel.

When the device is first powered up or the ResetDevice Method is called, the logical channel assignments revert to their defaults. Logical channel 0 is assigned to single-ended channel 0, logical channel 1 to single-ended channel 1 and so on.

The assignments made with this property are effective until they are either overwritten by setting the property again or the board is reset (hard reset or by calling the ResetDevice Method). There is no limitation on how often you may reassign logical channels.

The following constants are defined in the Eth32AnalogChannel enumerator. These are the valid physical channel sources to which a logical channel may be assigned. The constant definition should typically be used in your source code, but its hexadecimal value is shown for reference.

For single-ended channels, the reading comes from the voltage of the specified pin with respect to ground.

Table 7. Single-Ended Channels

Constant	Value	Physical Pin
ANALOG_SE0	&H00	Port 3, Bit 0
ANALOG_SE1	&H01	Port 3, Bit 1
ANALOG_SE2	&H02	Port 3, Bit 2
ANALOG_SE3	&H03	Port 3, Bit 3
ANALOG_SE4	&H04	Port 3, Bit 4
ANALOG_SE5	&H05	Port 3, Bit 5
ANALOG_SE6	&H06	Port 3, Bit 6
ANALOG_SE7	&H07	Port 3, Bit 7

For differential channels, the reading comes from the voltage difference between two pins. It is permissible for either to be positive or negative with respect to the other. They are simply labeled positive and negative inputs to specify how the reading is determined. Please note that the voltage on each pin must still remain within the range of 0 to 5V with respect to the ground of the device.

Table 8. Differential Channels

Constant	Value	Positive Input	Negative Input	Gain
ANALOG_DI00X10	&H08	Port 3, Bit 0	Port 3, Bit 0	10x
ANALOG_DI10X10	&H09	Port 3, Bit 1	Port 3, Bit 0	10x
ANALOG_DI00X200	&H0A	Port 3, Bit 0	Port 3, Bit 0	200x
ANALOG_DI10X200	&H0B	Port 3, Bit 1	Port 3, Bit 0	200x
ANALOG_DI22X10	&H0C	Port 3, Bit 2	Port 3, Bit 2	10x
ANALOG_DI32X10	&H0D	Port 3, Bit 3	Port 3, Bit 2	10x
ANALOG_DI22X200	&H0E	Port 3, Bit 2	Port 3, Bit 2	200x
ANALOG_DI32X200	&H0F	Port 3, Bit 3	Port 3, Bit 2	200x
ANALOG_DI01X1	&H10	Port 3, Bit 0	Port 3, Bit 1	1x
ANALOG_DI11X1	&H11	Port 3, Bit 1	Port 3, Bit 1	1x
ANALOG_DI21X1	&H12	Port 3, Bit 2	Port 3, Bit 1	1x
ANALOG_DI31X1	&H13	Port 3, Bit 3	Port 3, Bit 1	1x
ANALOG_DI41X1	&H14	Port 3, Bit 4	Port 3, Bit 1	1x
ANALOG_DI51X1	&H15	Port 3, Bit 5	Port 3, Bit 1	1x
ANALOG_DI61X1	&H16	Port 3, Bit 6	Port 3, Bit 1	1x
ANALOG_DI71X1	&H17	Port 3, Bit 7	Port 3, Bit 1	1x
ANALOG_DI02X1	&H18	Port 3, Bit 0	Port 3, Bit 2	1x
ANALOG_DI12X1	&H19	Port 3, Bit 1	Port 3, Bit 2	1x
ANALOG_DI22X1	&H1A	Port 3, Bit 2	Port 3, Bit 2	1x
ANALOG_DI32X1	&H1B	Port 3, Bit 3	Port 3, Bit 2	1x
ANALOG_DI42X1	&H1C	Port 3, Bit 4	Port 3, Bit 2	1x
ANALOG_DI52X1	&H1D	Port 3, Bit 5	Port 3, Bit 2	1x

Note that the entries above which show both the positive side and negative side with the same input pin can be used for calibration of the differential amplifier. Any nonzero reading from those indicates an offset error within the differential amplifier which you can subtract out of other channels that share the same negative input and gain.

Table 9. Calibration Reference Channels

Constant	Value	Description
ANALOG_122V	&H1E	Internal 1.22V Voltage Reference
ANALOG_0V	&H1F	0V (Ground)

The above two entries connect a logical channel to internal chip voltages. They can be used as calibration points to determine errors within the analog conversions.

Example

```
Private Sub example()

    ' Set up error handling for this routine
    On Error GoTo myerror

    Set dev = New Eth32

    ' .... Your code that establishes a connection here

    If dev.AnalogAssignment(0) = ANALOG_SE0 Then
        ' Logical channel 0 is configured for physical
        ' single-ended channel 0 (the default)
    End If

    ' Configure logical channel 7:  Assign it to the
    ' difference between bit 4 and bit 1 with 1X gain.
    dev.AnalogAssignment(7) = ANALOG_DI41X1


    Exit Sub
myerror:
    MsgBox "ETH32 error: " & dev.ErrorString(Err.Number)
End Sub
```

See Also

AnalogReference Property, AnalogState Property, InputAnalog Method

---

AnalogReference Property

```
Public Property AnalogReference As Eth32AnalogReference
```

Summary

This property configures the voltage source to be used by the Analog to Digital Converter as the reference voltage for analog conversions. The reference voltage determines the voltage level that will give the highest possible analog reading value. There are three possible voltages that may be used: An externally-generated voltage supplied on the analog reference pin, internal 5V, and internally generated 2.56V.

Parameters

This property does not have any parameters.

Value

This property is a Eth32AnalogReference enumerator type, which has the following valid values:

- REF_EXTERNAL - Selects the external, user-supplied voltage.
- REF_INTERNAL - Selects the internal 5V source.
- REF_256 - Selects the internal 2.56V reference.

Remarks

Note that whatever voltage source is selected will be internally connected to the external voltage reference pin. So for example, if you have a 4V source connected to the external reference pin, you should NOT configure the reference for REF_INTERNAL or REF_256 until you have disconnected the external reference pin.

Also note that if you connect a voltage to the external reference pin, it must not exceed 5V or go below 0V.

See Also

AnalogState Property, InputAnalog Method

AnalogState Property

```
Public Property AnalogState As Eth32AnalogState
```

Summary

This property enables or disables the Analog to Digital Converter (ADC) portion of the ETH32 device. The ADC must first be enabled before any valid analog readings can be obtained.

Parameters

This property does not have any parameters.

Value

This property is a Eth32AnalogState enumerator type, which has the following valid values:

- ADC_DISABLED - The Analog to Digital Converter is disabled. Analog readings will not be valid.

- ADC_ENABLED - The Analog to Digital Converter is enabled.

Remarks

Because the analog channels use the same physical pins as digital I/O port 3, enabling the ADC forces port 3 into input mode and sets the output value of port 3 to zero. Changes to the direction register or output value of port 3 are disabled while the ADC remains enabled. Note that regardless of what port 3's direction register and output value were at the time the ADC was enabled, if the ADC is later disabled, port 3 will be left in input mode with an output value of zero.

See Also

InputAnalog Method, AnalogAssignment Property, AnalogReference Property

---

CheckEvents Method

```
Public Sub CheckEvents()
```

Summary

This method forces the event queue to be checked for pending events. Any events will be immediately processed, causing your event handler routine to be called for each one. If this method is called when there are no pending events, it will simply return.

Parameters

This method does not have any parameters.

Return Value

This method does not return a value.

Remarks

In Visual Basic 6, your event handler function is called within the same thread as the rest of your application and events are processed around the same time that form events (such as Click) are processed. This means that processor-intensive portions of your code could potentially delay the processing of ETH32 events. In this case, if you would like to force any pending ETH32 events to be processed immediately, you may call this method.

Note that it is normally not necessary to ever use this method. Unless a situation like the above applies, incoming events will be processed automatically almost immediately after they occur.

See Also

Event Handler Section

Connect Method

```
Public Sub Connect(ByVal address As String, Optional ByVal port As Long = ETH32_PORT, _
                   Optional ByVal timeout As Long = 0)
```

Summary

The Connect method is used to open a connection to an ETH32 device. You must call Connect and successfully connect to an ETH32 device before calling other methods or accessing other properties of the Eth32 object. This method does NOT reset the device or change its configuration in any way.

Parameters

- address - The IP address or DNS name of the ETH32 device.
- port - The TCP port to connect to. If an overloaded method without this parameter is called, the constant ETH32_PORT (7152) is used, which is the port the ETH32 listens on.
- timeout - Specifies the maximum time, in milliseconds, that the connection attempt may take, excluding resolving DNS. You may specify a timeout of zero to use the default timeout from the system's TCP/IP stack, which is the default if this parameter is not specified. Note that the method may time out in less time than you specify if the system's timeout is shorter than what you specify. If the method does time out, it will raise an EthErrorTimeout error.

Return Value

This method does not have a return value. If any error occurs, an error will be raised.

Remarks

Once an object is connected to a device, you may not call Connect again on that object unless you first disconnect using the Disconnect Method. Note that your application may have connections open to several ETH32 devices at once. Each requires a separate Eth32 object to be created in your application.

Example

```
Private Sub example()

    ' Set up error handling for this routine
    On Error GoTo myerror

    Set dev = New Eth32

    ' NOTE: Substitute the IP address or DNS name of your device here.
    dev.Connect "192.168.1.100", ETH32_PORT, 10000

    ' Now that we're connected, turn on an LED:
    dev.Led(0) = True

    Exit Sub
myerror:
```

```
    If Err.Number = EthErrorTimeout Then
        MsgBox "Timed out while connecting to ETH32."
    Else
        MsgBox "Error connecting to ETH32: " & dev.ErrorString(Err.Number)
    End If
End Sub
```

See Also

Connected Property, Disconnect Method

Connected Property

```
Public Property Connected As Boolean
```

Summary

This is a read-only property that indicates whether the Connect Method has been successfully called on this object and that the Disconnect Method has not been called since then. Reading this property does not cause any communication with the device nor does it verify that the connection to the device is still good. For that, see the VerifyConnection Method.

If there is a connection to the device, this property will read as true. If there is not a connection to the device, rather than raising an error, this property will simply read false.

Parameters

This property does not have any parameters.

Value

This property is a boolean type. A true value means that the object is connected to an ETH32 device, while false means that it is not.

Example

```
Private Sub example()

    ' Set up error handling for this routine
    On Error GoTo myerror


    ' .... Your code here

    ' Assume that we don't know for sure whether the dev object
    ' is connected to a device, but that if it is, we want to
    ' disconnect it.  This code accomplishes that:
    If dev.Connected Then
        dev.Disconnect
    End If
```

```
    Exit Sub
myerror:
    MsgBox "ETH32 error: " & dev.ErrorString(Err.Number)
End Sub
```

See Also

Connect Method, Disconnect Method, VerifyConnection Method

ConnectionFlags Method

```
Public Function ConnectionFlags(ByVal reset As Long) As Eth32ConnectionFlag
```

Summary

The ETH32 device maintains several flag bits for each individual active TCP/IP connection. The flags indicate conditions that are (or were) present for that connection. Currently, these flags are used to indicate whether any data that needed to be sent to your application from the ETH32 device had to be discarded due to lack of queue space. This method retrieves the flags for this connection to the device. If instructed to do so, the method also clears all of the flags for this connection to zero immediately after retrieving them.

Parameters

- reset - If nonzero, specifies that the flags for this connection should be reset to zero immediately after retrieving them.

Return Value

This method returns a Eth32ConnectionFlag enumerator type. The return value may be made up of any combination (that is, a bitwise or) of the following enumerator flags. Each flag indicates which kind of data had to be discarded due to a full queue.

- CONN_FLAG_NONE - If the return value equals this exactly, then no flags were set.
- CONN_FLAG_RESPONSE - Response to a query for information (for example InputByte Method).
- CONN_FLAG_DIGITAL_EVENT - Digital event notification.
- CONN_FLAG_ANALOG_EVENT - Analog event notification.
- CONN_FLAG_COUNTER_EVENT - Counter event (rollover or threshold) notification.

Remarks

To understand the role of the connection flags, consider the following example. Suppose that digital events are enabled on port 0, bit 0 for your connection to the ETH32. Now suppose that port 0, bit 0 begins pulsing rapidly, generating a steady stream of event notifications. Finally, suppose that the connection to your application is having trouble (losing packets, etc). Due to the nature of TCP/IP, the event

notifications must be retained in the queue of the ETH32 device until a TCP/IP acknowledgement for them has been received from the PC (in case they need to be retransmitted). If the TCP/IP acknowledgements do not come promptly and the events keep occurring, the queue will eventually fill up and the ETH32 device will be forced to simply discard any new event notifications. Although this scenario is undesirable and should be avoided, if it does happen, it is helpful for your application to be able to detect that it happened and that data was lost. The flags keep track of this individually for each TCP/IP connection (that is, a full queue on one connection will not affect flags on another). Note that the flags are informational only - they do not affect the behavior of the device.

Once a flag is set, it will remain set until it is reset back to zero by passing a nonzero number to the *reset* parameter of this method. In this case, the flags will only be reset to zero if the connection has enough space to queue up the reply data. In other words, the flags will not be lost if the response itself is unable to be queued.

The connection flags for new connections always start out as zero. When the ResetDevice Method is called, the flags for the connection it was received on are cleared, but the flags for any other active connections are not affected.

Example

```
Private Sub example()
    Dim flags As Eth32ConnectionFlag

    ' Set up error handling for this routine
    On Error GoTo myerror

    Set dev = New Eth32

    ' .... Your code that establishes a connection here

    ' Retrieve the connection flags for this connection and
    ' simultaneously clear them to zero.
    flags = dev.ConnectionFlags(1)

    ' See which flags are set
    If flags And CONN_FLAG_RESPONSE Then
        ' The device ran out of queue space at some point
        ' when it was trying to respond to a query for information.
    End If

    If flags And CONN_FLAG_DIGITAL_EVENT Then
        ' Some digital event data was lost due to running out
        ' of queue space.
    End If

    ' and so on


    ' Or, to check whether any flags at all are set:
    If flags = CONN_FLAG_NONE Then
        ' No flags whatsoever are set
    Else
        ' At least one flag is set
    End If
```

```
    Exit Sub
myerror:
    MsgBox "ETH32 error: " & dev.ErrorString(Err.Number)
End Sub
```

See Also

VerifyConnection Method

CounterRollover Property

```
Public Property CounterRollover(ByVal counter As Long) As Long
```

Summary

This property defines the maximum permissible value for a counter. After the counter reaches the rollover value, the next count will cause the counter to be reset to 0 and a rollover event notification will be sent to any connections that have enabled that rollover event. For example, with a rollover threshold set to 35, the counter value will progress as follows: ..., 33, 34, 35, 0, 1, ... Because the comparisons and reset are done directly in hardware, no counts are missed during a rollover.

The valid range of the rollover threshold is from 0 to the maximum value of the counter (65535 for 16-bit counter 0, and 255 for 8-bit counter 1). The powerup default rollover threshold is 255 for 8-bit and 65535 for 16-bit counters.

Parameters

- counter - Specifies the counter number (0 or 1).

Value

This property is a Long. For counter 0 (a 16-bit counter), this may range from 0-65535. For counter 1 (an 8-bit counter), this may range from 0-255.

Remarks

There is one special case involving rollover thresholds. When the counter value is manually set to exactly the threshold value by writing to the CounterValue Property, the rollover will NOT occur and the rollover event will NOT fire on the next counter increment. Instead, the counter will increment past the threshold value. The event will not fire until the counter value has wrapped around and again exceeds the threshold. For example, suppose the rollover threshold is set to 10 on an 8-bit counter and the CounterValue Property is used to set the counter value to 10. As the input line pulses, the counter value would increment as follows: 11, 12, ..., 255, 0, 1, ..., 10, 0, 1, ..., 10, 0, ...

Please note that defining a rollover threshold with this property does not enable the current connection to actually receive the rollover event notifications when they occur. These must be enabled separately using the EnableEvent Method. Also note that rollover thresholds are common to all connections. Changing the

thresholds will affect other connections if they are utilizing that particular counter.

See Also

CounterState Property, CounterThreshold Property, EnableEvent Method

---

CounterState Property

```
Public Property CounterState(ByVal counter As Long) As Eth32CounterState
```

Summary

This property allows you to control or retrieve the state of the two counters on the ETH32 device. The counter state configures which input signal edge (rising or falling) will increment the counter value or whether the counter is disabled. Setting or accessing this property does not affect the current counter value in any way. For example, a counter that is disabled and then enabled again will retain its value.

Parameters

- int counter - Specifies the counter number (0 or 1).

Value

This property is a Eth32CounterState enumerator type, which has the following valid values:

- COUNTER_DISABLED - The counter is disabled. The counter value may still be accessed, but the counter will not increment as a result of input signals.
- COUNTER_FALLING - The counter will increment on the falling edge of the input signal.
- COUNTER_RISING - The counter will increment on the rising edge of the input signal.

See Also

CounterRollover Property, CounterValue Property

CounterThreshold Property

```
Public Property CounterThreshold(ByVal counter As Long) As Long
```

Summary

This property defines a counter event threshold that will cause an event to fire as the counter value passes the threshold. On the ETH32 device, only Counter 0 supports this (although both counters support rollover thresholds). An event is fired as a result of the counter surpassing the threshold, not meeting it. For example, with a threshold of 9, the counter's value would increment from 8 to 9 without firing the event, but it would fire as the counter incremented from 9 to 10. The valid range for a counter event threshold is from 0 to the maximum possible counter value (65535 for 16-bit counter 0). The powerup default

threshold is 0. The threshold has no other side-effects on the counter - it does not reset the counter to 0 like the rollover threshold.

Parameters

- counter - Specifies the counter number. This must be 0.

Value

This property is a Long. The valid range is from 0 to the maximum possible counter value (65535 for 16-bit counter 0).

Remarks

Please note that defining a threshold with this property does not enable the current connection to actually receive the event notifications when they occur. These must be enabled separately using the EnableEvent Method. Also note that event thresholds are common to all connections. Changing the thresholds will affect other connections if they are utilizing that particular counter event.

See Also

CounterState Property, CounterRollover Property, EnableEvent Method

---

CounterValue Property

```
Public Property CounterValue(ByVal counter As Long) As Long
```

Summary

This property allows you to read or write the current value of the counters on the ETH32 device. After you have enabled the counter with the CounterState Property, the value of the counter indicates how many times the counter has been incremented by the external counter input. This property can also be written in order to set the counter value, which can be useful for initializing the counter. All counters begin with a value of zero after powerup or reset.

Parameters

- counter - Specifies the counter number (0 or 1).

Value

This property is a Long. For counter 0 (a 16-bit counter), this may range from 0-65535. For counter 1 (an 8-bit counter), this may range from 0-255.

See Also

CounterState Property, CounterRollover Property

DisableEvent Method

```
Public Sub DisableEvent(ByVal eventtype As Eth32EventType, ByVal port As Long, _
                        ByVal bit As Long)
```

Summary

This method instructs the ETH32 device to stop sending event notifications for the specified event on this connection to the device. It performs the opposite task of the EnableEvent Method.

Parameters

- eventtype - The type of event to disable. This parameter is a Eth32EventType enumerator type, which has the following valid values:
  - EVENT_DIGITAL - Digital I/O event. This includes port events and bit events.
  - EVENT_ANALOG - Analog event based on thresholds defined with the SetAnalogEventDef Method.
  - EVENT_COUNTER_ROLLOVER - Counter rollover event, which occurs when the counter rolls over to zero.
  - EVENT_COUNTER_THRESHOLD - Counter threshold event, which occurs when the counter passes a threshold defined with the CounterThreshold Property.
  - EVENT_HEARTBEAT - Periodic event sent by the device to indicate the TCP/IP connection is still good.
- port - For digital events, specifies the port number, for analog events, specifies the bank number, and for either counter event, specifies the counter number.
- bit - For digital events, this should be -1 for port events or the bit number (0-7) for bit events. For analog events, this specifies the analog channel number (0-7).

Return Value

This method does not return a value.

See Also

EnableEvent Method

Disconnect Method

```
Public Sub Disconnect()
```

Summary

This method closes the connection to the ETH32 device and cleans up all of the resources within the API that were used for the connection. After this method returns, most of the methods and properties of the object won't be able to be successfully used until another connection has been formed using the Connect Method.

Parameters

This method does not have any parameters.

Return Value

This method does not return a value.

Remarks

You should be careful to always call this method when you are finished using the device. The device has a limited number of connections it can support and if you do not disconnect and your application continues executing, you will continue using one of those connections. If you fail to call this method, your connections will remain open potentially until your application terminates.

In this Visual Basic 6 class, you also must be particularly careful to call this method when your application is shutting down. If your application is structured to exit when the last form has been closed (as opposed to using an End statement), then any open ETH32 connection will prevent your application from actually closing. This is because each connected Eth32 class uses a hidden form to assist in processing events from the device. If Disconnect hasn't been called, this form will still exist, preventing the application from exiting. Because the form is hidden, you may not even realize your application is still running unless you look closely in the task manager.

To summarize: Always call Disconnect. Don't depend on application cleanup to do it for you.

It is a good idea to put code similar to the following in the Form_Unload event of your main form. This code assumes that your object variable name is *dev*:

```
' When this form unloads, make sure the connection is closed, otherwise
' it will keep the application running.
If Not (dev Is Nothing) Then
        ' dev is at least instantiated
        If dev.Connected Then
                dev.Disconnect
        End If
End If
```

See Also

Connect Method, Connected Property

EmptyEventQueue Method

```
Public Sub EmptyEventQueue()
```

Summary

This method empties the event queue within the API. This method does not have an effect on the ETH32 device itself.

Parameters

This method does not have any parameters.

Return Value

This method does not return a value.

See Also

EventQueueCurrentSize Property, EventQueueLimit Property

EnableEvent Method

```
Public Sub EnableEvent(ByVal eventtype As Eth32EventType, ByVal port As Long, _
                       ByVal bit As Long, ByVal id As Long)
```

Summary

This method enables reception of the specified event on this connection to the device. The ETH32 device only sends event notifications to those connections that specifically request them, so this method requests notification for the specified event from the device, as well as internally assigns the event an ID number provided by you.

Parameters

- eventtype - The type of event to enable. This parameter is a Eth32EventType enumerator type, which has the following valid values:
  - EVENT_DIGITAL - Digital I/O event. This includes port events and bit events.
  - EVENT_ANALOG - Analog event based on thresholds defined with the SetAnalogEventDef Method.

  - EVENT_COUNTER_ROLLOVER - Counter rollover event, which occurs when the counter rolls over to zero.
  - EVENT_COUNTER_THRESHOLD - Counter threshold event, which occurs when the counter passes a threshold defined with the CounterThreshold Property.
  - EVENT_HEARTBEAT - Periodic event sent by the device to indicate the TCP/IP connection is still good.

- port - For digital events, specifies the port number, for analog events, specifies the bank number, and for either counter event, specifies the counter number.
- bit - For digital events, this should be -1 for port events or the bit number (0-7) for bit events. For analog events, this specifies the analog channel number (0-7).
- id - You may specify any number to be associated with this event.

Return Value

This method does not return a value.

Remarks

The *id* parameter allows you to assign any arbitrary number to this particular event. The ID you assign is included with the event information whenever this event fires. The idea is that you can identify a particular event with a single comparison rather than needing to inspect several pieces of data such as the event type, port number, and bit number. The ID number is completely arbitrary and multiple events may be given the same ID number if desired. The ID numbers are stored within the API and are not sent to the ETH32 device.

One other minor technicality is that the heartbeat event is permanently enabled on the ETH32 device itself for purposes of connection maintenance. Therefore, for the heartbeat event, this method simply enables the event within the API, meaning that when the event comes in, rather than being discarded it will be added to the event queue. The one small side-effect to this fact is that if you have enabled reception of the heartbeat event and another connection calls the ResetDevice Method, you will continue to receive heartbeat events, whereas all other event types will have been disabled on the device itself. Note that if you call ResetDevice on your own connection, it automatically disables the heartbeat event within the API for your connection, so in that case it is not an issue.

Example

This example is a very simple, yet compilable, example of how to utilize events. To compile this example, create a new project, add the Eth32 support files as described in the Getting Started section, and create a button named setup_button on the main form. Then, paste this code into the code window for that form:

```
Option Explicit
Dim WithEvents dev As Eth32


Private Sub dev_EventFired(ByVal id As Long, ByVal eventtype As Long, ByVal port As Long, _
```

```
                            ByVal bit As Long, ByVal prev_value As Long, ByVal value As Long, _
                            ByVal direction As Long)

    MsgBox "An event has fired.  ID: " & id & " Value: " & value

End Sub

Private Sub Form_Unload(Cancel As Integer)

    ' When this form unloads, make sure the connection is closed, otherwise
    ' it will keep the application running.
    If Not (dev Is Nothing) Then
        ' dev is at least instantiated
        If dev.Connected Then
            dev.Disconnect
        End If
    End If

End Sub

Private Sub setup_button_Click()
    ' Assume this button is clicked by the user when he wants to connect to the
    ' device and configure event handling

    Set dev = New Eth32

    ' Set up error handling for this routine
    On Error GoTo myerror

    ' NOTE: Substitute the IP address or DNS name of your device here.
    dev.Connect "192.168.1.100"

    ' If there is a pushbutton connected between Port 0, bit 0 and ground,
    ' then we can provide an internal pullup causing it to float high by
    ' doing:
    dev.OutputBit 0, 0, 1

    ' Look for events on Port 0, bit 0.
    dev.EnableEvent EVENT_DIGITAL, 0, 0, 100
    Exit Sub
myerror:
    MsgBox "Error communicating with the ETH32: " & dev.ErrorString(Err.Number)

End Sub
```

See Also

Event Handler Section, DisableEvent Method

ErrorString Method

```
Public Function ErrorString(errorcode As EthError) As String
```

Summary

This method translates an error code into a string which briefly describes the error. It is not necessary to have a connection to the ETH32 device in order to use this method.

Parameters

- errorcode - The error code to translate into a string. This parameter is a EthError enumerator type. Possible error codes are listed in the Error Codes section.

Return Value

This method returns a string, which provides a brief description of the given error code.

Example

```
Private Sub example()

    ' Set up error handling for this routine
    On Error GoTo myerror

    Set dev = New Eth32

    ' .... Your code that establishes a connection here

        ' .... More of your code that performs operations on the device or other things.

    Exit Sub
myerror:
    MsgBox "ETH32 error: " & dev.ErrorString(Err.Number)
End Sub
```

See Also

Error Handling Section

EventQueueCurrentSize Property

```
Public Property EventQueueCurrentSize As Long
```

Summary

This read-only property allows you to determine how many events are currently in the event queue within the API. This property does not communicate with the ETH32 device or provide information about the device itself. For more information about the API event queue, see the EventQueueLimit Property.

Parameters

This property does not have any parameters.

Value

This property is a Long. The value of the property is the number of events currently waiting in the API's event queue.

See Also

EmptyEventQueue Method, EventQueueLimit Property

EventQueueLimit Property

```
Public Property EventQueueLimit As Long
```

Summary

This property controls the maximum allowable size of the event queue within the API. If a nonzero maximum size is configured for the event queue (which is the default when a new connection is created), the API will enable events and queue any events that arrive while your event handler function is already busy processing an event. If a zero maximum size is configured, event processing will be disabled. This property only controls the behavior of the API. It does not affect anything on the actual ETH32 device.

Parameters

This property does not have any parameters.

Value

This property is a Long. Its value specifies the maximum number of events that are allowed to be queued by the API.

Remarks

Your event handler routine is called once for each event notification that is sent by the device. Events are processed one at a time and in the sequence that they are sent by the device. The event queue is used to store events that have arrived, but have not yet been sent to your event handler routine. This is particularly important if your event handler routine takes a significant time to execute.

If the event queue ever becomes full and more events arrive, the behavior of the API will depend on the current setting of the EventQueueMode Property.

Example

```
Private Sub example()

    ' Set up error handling for this routine
    On Error GoTo myerror

    Set dev = New Eth32

    ' .... Your code that establishes a connection here

    ' Configure the event queue to hold up to 1,000 events.
    ' If the queue is ever full and more events arrive, discard
    ' the new events.
    dev.EventQueueLimit = 1000
    dev.EventQueueMode = QUEUE_DISCARD_NEW

    Exit Sub
myerror:
    MsgBox "ETH32 error: " & dev.ErrorString(Err.Number)
End Sub
```

See Also

EnableEvent Method, EventQueueCurrentSize Property, EventQueueMode Property

---

EventQueueMode Property

```
Public Property EventQueueMode As Eth32QueueMode
```

Summary

This property configures the behavior of the event queue within the API. If the event queue ever becomes full (reaches the limit configured by the EventQueueLimit Property) and new events arrive, either old events will be shifted out to make room for the new, or the new events will be ignored, depending on the behavior you have specified with this property. The QUEUE_DISCARD_NEW setting is the default when a new connection is created.

Parameters

This property does not have any parameters.

Value

This property is a Eth32QueueMode enumerator type, which has the following valid values:

- QUEUE_DISCARD_NEW - When the queue is full, discard any new events.
- QUEUE_DISCARD_OLD - When the queue is full, shift out the oldest event to make room for the new event at the end of the queue.

Remarks

The event queue size that is considered full is defined by the EventQueueLimit Property.

See Also

EventQueueLimit Property

FirmwareMajor Property

```
Public Property FirmwareMajor As Long
```

Summary

This read-only property retrieves the "major" portion of the firmware version number from the device. The firmware version consists of a major number and minor number. When displayed as a string, it is typically formatted as major.minor with minor zero-padded to three digits if necessary. For example, for release 2.001, the major number is 2 and the minor number is 1.

Parameters

This property does not have any parameters.

Value

This property is a Long. Its value is the major number of the firmware version.

See Also

FirmwareMinor Property

FirmwareMinor Property

```
Public Property FirmwareMinor As Long
```

Summary

This read-only property retrieves the "minor" portion of the firmware version number from the device. The firmware version consists of a major number and minor number. When displayed as a string, it is typically formatted as major.minor with minor zero-padded to three digits if necessary. For example, for release 2.001, the major number is 2 and the minor number is 1.

Parameters

This property does not have any parameters.

Value

This property is a Long. Its value is the minor number of the firmware version.

See Also

FirmwareMajor Property

GetAnalogEventDef Method

```
Public Sub GetAnalogEventDef(ByVal bank As Long, ByVal channel As Long, _
                             ByRef lomark As Long, ByRef himark As Long)
```

Summary

This method retrieves the low and high thresholds defined for the specified analog event bank and channel. Please see the SetAnalogEventDef Method for more information about the analog event definition and thresholds.

Parameters

- bank - Identifies which bank of analog events from which to retrieve information (0 or 1).
- channel - Identifies the analog channel (0-7).
- lomark - Output parameter which will receive the low threshold (8-bit value) for the analog event.
- himark - Output parameter which will receive the high threshold (8-bit value) for the analog event.

Return Value

This method does not return a value.

Remarks

Note that this method does not retrieve the default value that was specified when the thresholds were set. This is because the default value is only used during the moment that the thresholds are defined and is not applicable after that point.

See Also

SetAnalogEventDef Method

GetDirection Method

```
Public Function GetDirection(ByVal port As Long) As Long
```

Summary

This method retrieves the current direction register for the specified digital I/O port. See the SetDirection Method for further description of the direction register.

Parameters

- port - The port number (0-5).

Return Value

This method returns a Long. The return value is the port's current direction register.

See Also

GetDirectionBit Method, SetDirection Method, SetDirectionBit Method

---

GetDirectionBit Method

```
Public Function GetDirectionBit(ByVal port As Long, ByVal bit As Long) As Long
```

Summary

This method retrieves the value of a single bit of a port's direction register. It is provided simply for convenience, since it internally calls the GetDirection Method to determine the value of the specified bit.

Parameters

- port - Specifies the port number (0-5).
- bit - Specifies the bit number (0-7) within the port.

Return Value

This method returns a Long. The return value is the value of the specified direction bit of the specified port.

See Also

GetDirection Method, SetDirection Method, SetDirectionBit Method

---

GetEeprom Method

```
Function GetEeprom(address As Long, length As Long) As Byte()
```

Summary

This method retrieves data from the non-volatile EEPROM memory of the device.

Parameters

- address - The starting location from which data should be retrieved (0-255).
- length - The number of bytes to retrieve. Valid values for this parameter depend on what is provided for the address parameter. For example, with an address of 0, you may specify a length of all 256 bytes, but with an address of 255, length may only be 1.

Return Value

This method returns a byte array containing the requested data.

See Also

SetEeprom Method

GetPwmParameters Method

```
Public Sub GetPwmParameters(ByVal channel As Long, ByRef state As Eth32PwmChannel, _
                            ByRef freq As Single, ByRef duty As Single)
```

Summary

This method is provided for your convenience in working with all of the various PWM settings. It internally calls several of the other API functions to determine the current state of the specified PWM channel and calculate its configuration in more familiar terms (hertz and percentage). This method calculates the frequency and duty cycle of the channel from the PWM base period and the channel's duty period.

Parameters

- channel - Specifies the PWM channel number (0 or 1).
- state - Output parameter which will receive the current state of the PWM channel. This will be one of the following values of the Eth32PwmChannel enumerator:
  - PWM_CHANNEL_DISABLED - The PWM pin is configured as a normal digital I/O pin.
  - PWM_CHANNEL_NORMAL - The PWM pin is configured as a PWM output. It will be high for the time specified by the duty period and low for the rest of the PWM base period.

  - PWM_CHANNEL_INVERTED - The PWM pin is configured as a PWM output. It will be low for the time specified by the duty period and high for the rest of the PWM base period.

- freq - Output parameter which will receive the current frequency of the PWM channels in Hertz.
- duty - Output parameter which will receive the duty cycle of the PWM channel. This may range from 0.00 to 1.00, representing the duty cycle as a percentage.

Return Value

This method does not return a value.

See Also

SetPwmParameters Method

InputAnalog Method

```
Public Function InputAnalog(ByVal channel As Long) As Long
```

Summary

This method retrieves an analog reading from one of the analog channels on the device. The analog readings are only meaningful when the ADC has been enabled (see the AnalogState Property). The analog readings are 10-bit values. See below for further explanation of their meaning.

Parameters

- channel - Specifies the logical analog channel (0-7) to read. Note that each logical analog channel may be arbitrarily assigned to physical channels using the AnalogAssignment Property.

Return Value

This method returns a Long. The return value is the reading from the specified channel.

Remarks

The reading that is obtained with this method is a 10-bit value (range of 0-1023) representing the voltage level relative to the analog reference voltage. The exact interpretation depends on whether a single-ended or differential channel has been selected (see the AnalogAssignment Property).

For single-ended channels, the reading is:

```
(analog reading) = (channel voltage * 1024) / (voltage reference)
```

For example, a reading of 0 means 0V and a reading of 1023 means a voltage just under the voltage reference (assuming internal 5V reference, about 4.99V). Once you have the analog reading, you can calculate the input voltage that produced it by calculating:

```
voltage = (analog reading)/1024 * (voltage reference)
```

For differential channels, the reading is:

```
(analog reading) = 512 + (positive side voltage - negative side voltage) * GAIN * 512 / (voltage reference)
```

For example, assuming a gain of 1X, a reading of 0 means the positive pin is (voltage reference) volts less than the negative pin, a reading of 512 means the positive pin and negative pin are at the same voltage, and a reading of 1023 means the positive pin is almost (voltage reference) volts higher than the negative pin. Once you have the analog reading, you can calculate the voltage of the positive pin relative to the negative pin by calculating:

```
voltage = (analog reading - 512) / 512 * (voltage reference)
```

Example

```
Private Sub example()
    Dim chan0 As Long
    Dim voltage As Double

    ’ Set up error handling for this routine
    On Error GoTo myerror

    Set dev = New Eth32

    ’ .... Your code that establishes a connection here

    ’ Enable the Analog to Digital Converter
    dev.AnalogState = ADC_ENABLED

    ’ Configure logical channel 0 to read the physical channel 0 relative to ground (single-ended)
    ’ This is the power-on default anyway, but is shown here for completeness:
    dev.AnalogAssignment(0) = ANALOG_SE0

    ’ Configure the analog voltage reference to be the internal 5V source
    dev.AnalogReference = REF_INTERNAL

    ’ Finally, read the voltage on channel 0
    chan0 = dev.InputAnalog(0)

    ’ Now, determine whether the voltage was >= 3V.  Remember
    ’ we’re using a 5V voltage reference.
    If chan0 >= (3# / 5# * 1024) Then
        ’ The voltage on channel 0 was at least 3V
    Else
        ’ The voltage was less than 3V
    End If

    ’ If you want to calculate the voltage:
    voltage = chan0 / 1024# * 5#

    Exit Sub
myerror:
    MsgBox "ETH32 error: " & dev.ErrorString(Err.Number)
End Sub
```

See Also

AnalogAssignment Property, AnalogReference Property, AnalogState Property

---

InputBit Method

```
Public Function InputBit(ByVal port As Long, ByVal bit As Long) As Long
```

Summary

This method retrieves the value of a single bit within a digital I/O port. It is provided simply for convenience, since it internally calls the InputByte Method to determine the value of the specified bit.

Parameters

- port - Specifies the port number (0-5) to read.
- bit - Specifies the bit number (0-7).

Return Value

This method returns a Long. The return value is the current value (0 or 1) of the specified bit.

See Also

InputByte Method, OutputBit Method, SetDirectionBit Method

InputByte Method

```
Public Function InputByte(ByVal port As Long) As Long
```

Summary

This method retrieves the current input value of the specified digital I/O port on the device. When a port is configured as an input port (using the SetDirection Method), the input value represents the voltage levels on the port's pins. For each bit, a low voltage (close to 0V) yields a 0-bit in the input value and a high voltage (close to 5V) yields a 1-bit.

Parameters

- port - Specifies the port number (0-5) to read.

Return Value

This method returns a Long. The return value is the current input value of the specified port.

Example

```
Private Sub example()
    Dim portval As Long

    ' Set up error handling for this routine
    On Error GoTo myerror

    Set dev = New Eth32

    ' .... Your code that establishes a connection here

    ' Read the input value of port 2
    portval = dev.InputByte(2)

    ' See whether any of bits 0-3 are high (1)
    If (portval And &H0F) <> 0 Then
        ' At least one of bits 0-3 are high
    Else
        ' None of bits 0-3 are high
    End If



    Exit Sub
myerror:
    MsgBox "ETH32 error: " & dev.ErrorString(Err.Number)
End Sub
```

See Also

InputBit Method, OutputByte Method, SetDirection Method

InputSuccessive Method

```
Public Function InputSuccessive(ByVal port As Long, ByVal maxcount As Long, _
                                ByRef status As Long) As Long
```

Summary

This method instructs the ETH32 device to read the specified port multiple times in succession until two consecutive reads yield the same result. This method is useful for situations where a multi-bit value is being read, for example, the output of a digital counter chip. When reading such a value, it is always possible to read the value during a transition state as bits are changing and an invalid value is represented. By requiring that two successive reads match, any invalid transition values are automatically ignored. The device continues to read the port until one of the following conditions is met:

- Two successive (in other words, back to back) reads give the same port value. This value is returned.
- The port was read the maximum number of times specified in the command without a match occurring.

This functionality is implemented directly within the ETH32 device (as opposed to the API), making it very fast and efficient with network traffic.

Parameters

- port - Specifies the port number (0-3) to read.
- maxcount - The maximum number of times to read the port (2-255).
- status - Output parameter which will receive the number of times the port had to be read to get a successive match. If no match was ever seen, this will be zero.

Return Value

This method returns a Long. The return value is the last value read from the port, regardless of whether or not two successive reads ever matched.

Example

```
Private Sub example()
    Dim portval As Long
    Dim status As Long

    ' Set up error handling for this routine
    On Error GoTo myerror

    Set dev = New Eth32

    ' .... Your code that establishes a connection here

    ' Read the value of an 8-bit counter on port 0, limit to 20 reads
    portval = dev.InputSuccessive(0, 20, status)

    If status = 0 Then
        ' Never saw the same value twice in a row
    Else
        ' The port value is in the portval variable
    End If

    Exit Sub
myerror:
    MsgBox "ETH32 error: " & dev.ErrorString(Err.Number)
End Sub
```

See Also

InputByte Method, SetDirection Method

Led Property

```
Public Property Led(ByVal lednum As Long) As Boolean
```

Summary

This property allows you to control or retrieve the state of the two LED's built into the ETH32 device.

Parameters

- lednum - Identifies the LED (0 or 1) to control or inspect.

Value

This property is a boolean type. A true value means the LED is on and a false value means the LED is off.

Example

```
Private Sub example()

    ' Set up error handling for this routine
    On Error GoTo myerror

    Set dev = New Eth32

    ' .... Your code that establishes a connection here

    ' Determine whether LED 0 is on or off
    If dev.Led(0) Then
        ' LED is on
    Else
        ' LED is off
    End If

    ' Turn on LED 1
    dev.Led(1) = True

    Exit Sub
myerror:
    MsgBox "ETH32 error: " & dev.ErrorString(Err.Number)
End Sub
```

OutputBit Method

```
Public Sub OutputBit(ByVal port As Long, ByVal bit As Long, ByVal value As Long)
```

Summary

This method alters a single bit of the output value of any I/O port without affecting the value of any other bits. See the OutputByte Method for further description of the output value.

Parameters

- port - The port number (0-5).
- bit - The bit number (0-7).
- val - Any nonzero number sets the bit to 1 and zero clears the bit to 0.

Return Value

This method does not return a value.

Remarks

This method alters the specified bit's value in a single operation directly on the ETH32 device. In other words, it does NOT read the current value over the network, modify it and then write it back. By doing it in a single operation, this avoids the potential of inadvertently overwriting changes made to other bits by other connections.

Port 3 shares its pins with the analog channels. When the ADC is enabled, all pins of port 3 are forced into input mode and the output value is set to zero. Port 3's output value cannot be modified while the ADC is enabled.

See Also

InputBit Method, OutputByte Method, SetDirectionBit Method

OutputByte Method

```
Public Sub OutputByte(ByVal port As Long, ByVal value As Long)
```

Summary

This method writes a new output value to one of the digital I/O ports on the device. When the port is configured as an output port (using the SetDirection Method), each bit of the output value determines the voltage (0 or 5V) of the corresponding bit of the port. When the port is configured as an input port, any 1-bits in the output value enables a weak pullup for that bit of the port.

Parameters

- port - The port number to write to (0-5).
- value - The new value for the port. This may be 0-255 for ports 0-3 and 0-1 for the single-bit ports 4 and 5.

Return Value

This method does not return a value.

Remarks

Port 3 shares its pins with the analog channels. When the ADC is enabled, all pins of port 3 are forced into input mode and the output value is set to zero. Port 3's output value cannot be modified while the ADC is enabled.

Example

```
Private Sub example()

    ' Set up error handling for this routine
    On Error GoTo myerror

    Set dev = New Eth32

    ' .... Your code that establishes a connection here


    ' Set port 0 pins to be outputs
    dev.SetDirection 0, 255

    ' Write a new value for port 0
    dev.OutputByte 0, 85


    Exit Sub
myerror:
    MsgBox "ETH32 error: " & dev.ErrorString(Err.Number)
End Sub
```

See Also

InputByte Method, OutputBit Method, Readback Method, SetDirection Method

---

ProductID Property

```
Public Property ProductID As Long
```

Summary

This read-only property retrieves the product ID from the device, which identifies the type/model of the device.

Parameters

This property does not have any parameters.

Value

This property is a Long. Its value is a numeric code representing the type or model of the device.

See Also

SerialNum Property

PulseBit Method

```
Public Sub PulseBit(ByVal port As Long, ByVal bit As Long, ByVal edge As Eth32PulseEdge, _
                    ByVal count As Long)
```

Summary

This method outputs a burst of pulses on the port and bit specified. This can be useful, for example, in quickly clocking an external digital counter a specified number of times. You should ensure that the specified bit is configured as an output bit before calling this method. The ETH32 device implements the pulse functionality (as opposed to the API), which means it is performed very quickly and is efficient for network traffic.

Parameters

- port - The port number (0-5).
- bit - The bit number (0-7) on the specified port that should be pulsed.
- edge - Specifies whether the pulses should be falling or rising edge. This parameter is a Eth32PulseEdge enumerator type, which has the following valid values:
    - PULSE_FALLING - Bit is set low, then high, for each pulse.
    - PULSE_RISING - Bit is set high, then low, for each pulse.
- count - The number of times to pulse the bit. May be up to 255.

Return Value

This method does not return a value.

Remarks

The falling edge mode would typically be used on a bit that is initially high (and likewise rising edge with low), but this is not required. If a single falling edge pulse is performed on a bit that is already low, the pulse will end up simply setting the bit high. The reverse applies to a rising edge pulse where the bit is already high.

See Also

OutputBit Method, SetDirectionBit Method

PwmBasePeriod Property

```
Public Property PwmBasePeriod As Long
```

Summary

This property configures the main PWM clock to have a cycle period of the given number of counts. This defines the base frequency that will be used for the PWM channels. The base frequency is not individually selectable for each channel, so this property affects both PWM outputs. Each complete PWM waveform will have a duration of (BasePeriod + 1) PWM clock cycles. The PWM clock counts at a rate of 2 MHZ. This means, for example, that specifying a period of 99 would result in a frequency of 20 KHZ (2,000,000/(99+1)). The base period is specified as a 16-bit number that may range from a value of 49 (40 KHZ) to a value of 65,535 (30.5 HZ).

Parameters

This property does not have any parameters.

Value

This property is a Long. It specifies the number of PWM clock counts that make up the base period of the PWM channels. This may range from 49 - 65535.

See Also

PwmChannel Property, PwmClockState Property, PwmDutyPeriod Property, SetPwmParameters Method

---

PwmChannel Property

```
Public Property PwmChannel(ByVal channel As Long) As Eth32PwmChannel
```

Summary

This property configures the state of the PWM channels. When a channel is disabled, the I/O pin will function as a normal digital I/O pin. When the channel is enabled, that I/O pin will be overridden and the pin will become the PWM output. However, note that the pin must be put into output mode using the SetDirection Method or SetDirectionBit Method.

Parameters

- channel - Specifies the PWM channel number whose state should be set (0 or 1).

Value

This property is a Eth32PwmChannel enumerator type, which has the following valid values:

- PWM_CHANNEL_DISABLED - The PWM pin will function as a normal digital I/O pin.
- PWM_CHANNEL_NORMAL - The PWM pin will function as a PWM output. It will be high for the time specified by the duty period and low for the rest of the PWM base period.
- PWM_CHANNEL_INVERTED - The PWM pin will function as a PWM output. It will be low for the time specified by the duty period and high for the rest of the PWM base period.

See Also

PwmBasePeriod Property, PwmClockState Property, PwmDutyPeriod Property, SetPwmParameters Method

---

PwmClockState Property

```
Public Property PwmClockState As Eth32PwmClock
```

Summary

This property enables or disables the PWM clock from counting. The PWM clock is shared between both PWM outputs of the device. When the PWM clock is disabled, the PWM outputs will be idle (not pulsing). The PWM clock may be enabled or disabled independently of whether the individual PWM channel outputs are enabled or disabled.

Parameters

This property does not have any parameters.

Value

This property is a Eth32PwmClock enumerator type, which has the following valid values:

- PWM_CLOCK_DISABLED - Disables the PWM clock.
- PWM_CLOCK_ENABLED - Enables the PWM clock.

See Also

PwmBasePeriod Property, PwmChannel Property, PwmDutyPeriod Property, SetPwmParameters Method

---

PwmDutyPeriod Property

```
Public Property PwmDutyPeriod(ByVal channel As Long) As Long
```

Summary

This property defines the duty period for a PWM channel, which is the length of time the PWM output is active during each PWM cycle. The duty period is specified as PWM clock counts less one. In other words, when the PWM channel state is in normal mode, the PWM output will be high for (DutyPeriod + 1) counts of the PWM clock and low for the remainder of the clock counts in the cycle. The length of the PWM cycle is called the base period and set using the PwmBasePeriod Property.

Parameters

- channel - Specifies the PWM channel number (0 or 1).

Value

This property is a Long. The value specifies the duty period for the channel in terms of PWM clock counts. The valid range is from 0-65535.

Remarks

Remember that the base period (set with the PwmBasePeriod Property) is shared between both PWM channels on the device. However, the duty period (set with this property) is individually configurable for each channel. The recommended approach is to choose a PWM frequency that is appropriate for both channels and set the base period accordingly once during initialization. After that point, the individual duty periods for each channel should be set whenever necessary in order to alter the percentage of time the channel is on (duty cycle).

Any 16-bit value can be specified for the period, from 0 to 65535. Note that if a duty period is given that is greater than or equal to the current PWM base period, the output will be a solid high (in normal mode) or a solid low (in inverted mode). If a duty period of 0 is given, the output will not be solid, but rather it will have a short spike during each period of the PWM clock.

Example

```
Private Sub example()

    ' Set up error handling for this routine
    On Error GoTo myerror

    Set dev = New Eth32

    ' .... Your code that establishes a connection here
```

```
    ' Set up PWM channel 0 to have a 10 KHZ, 60% PWM signal:

    ' First, set up the base period to give a frequency of 10 KHZ
    ' This is calculated as:
    '   (2,000,000)/(10,000) - 1
    ' We subtracted one since the base period takes one clock
    ' cycle longer than the value we load in.
    dev.PwmBasePeriod = 199

    ' Set up this PWM channel's duty period to take up 60% of
    ' each base period cycle.  The base period takes 200 clock
    ' cycles, so we want the duty period to take:
    '   200 * 0.60 = 120 clock cycles
    ' Since the duty period takes one cycle longer than the value
    ' we load into it, we specify 119 here:
    dev.PwmDutyPeriod(0) = 119


    ' Put the PWM pin into output mode
    ' PWM 0's output pin is on Port 2, bit 4
    dev.SetDirectionBit 2, 4, 1

    ' Enable the main PWM clock
    dev.PwmClockState = PWM_CLOCK_ENABLED

    ' Finally, enable the PWM channel
    dev.PwmChannel(0) = PWM_CHANNEL_NORMAL

    Exit Sub
myerror:
    MsgBox "ETH32 error: " & dev.ErrorString(Err.Number)
End Sub
```

See Also

PwmBasePeriod Property, PwmChannel Property, SetPwmParameters Method

---

Readback Method

```
Public Function Readback(ByVal port As Long) As Long
```

Summary

This method retrieves (reads back) the current output value for the specified port. This is the value that was last written by calling the OutputByte Method or one or more calls to the OutputBit Method.

Parameters

- port - The port number to read back (0-5)

Return Value

This method returns a Long. The return value is the port's current output value.

See Also

OutputBit Method, OutputByte Method

ResetDevice Method

```
Public Sub ResetDevice()
```

Summary

This method resets most aspects of the device to their power-up default status. It does not perform a "cold reset" of the device. All TCP/IP connections to the device are preserved and do not need to be reestablished. See the remarks below for a list of everything that is affected.

Parameters

This method does not have any parameters.

Return Value

This method does not have a return value.

Remarks

The following parts of the device are reset by this method

- All digital I/O ports are configured as inputs.
- The output values of all digital I/O ports are set to zero.
- The LED's are turned off
- The Analog to Digital Converter is disabled.
- The analog voltage reference is configured to the external reference (REF_EXTERNAL).
- The analog channel assignments are all set to the single-ended channels. Logical channel 0 is set to single-ended channel 0, Logical channel 1 to single-ended 1, and so on.
- All events are disabled for all connections.
- Analog event definitions are cleared.

- Both counters are disabled.
- Counter values are set to zero.
- Counter rollover points are set to their highest possible values (&HFFFF for 16-bit counter 0, &HFF for 8-bit counter 1).
- Counter event threshold (applies only to counter 0) set to zero.
- PWM channels are disabled and the main PWM clock is stopped.
- The PWM base period is set to its highest (lowest frequency) setting of &HFFFF counts.
- The duty period of both PWM channels is set to zero.
- The connection flags are reset only for the connection that performed the reset. The connection flags for any other connections are not affected.

SerialNum Property

```
Public Property SerialNum As String
```

Summary

This read-only property retrieves the serial number of the ETH32 device in string format as it is printed on the device.

Parameters

This property does not have any parameters.

Value

This property is a string. Its value is the string representation of the device's serial number.

Remarks

The serial number is made up of several components and arranged as follows:

```
(productid)-(batch)(unit)
```

where:

- productid is a number identifying the product type/model. This number is returned by the ProductID Property.
- batch is the batch number formatted as two letters. 1 becomes AA, 2 becomes AB, etc.

- unit is the unit number, zero padded to 3 digits if necessary.

See Also

ProductID Property

SetAnalogEventDef Method

```
Public Sub SetAnalogEventDef(ByVal bank As Long, ByVal channel As Long, ByVal lomark As Long, _
                             ByVal himark As Long, ByVal defaultval As Eth32AnalogEvtDef)
```

Summary

This method defines the event thresholds for a single logical analog channel in the specified analog event bank. The thresholds that are defined determine what analog readings will cause the event to fire. The thresholds allow the event logic on the ETH32 device to assign a current state (high or low) to the event. The event will be considered high if the analog reading is at or above the given hi-mark and will be considered low if at or below the given lo-mark. Whenever the state of the event changes (low to high or high to low), an event notification will be fired. When the analog reading is between the lo-mark and hi-mark, it will retain its previous value. This allows "hysteresis" to be built into the event so that a fluctuating signal will not cause an event to continuously fire. The thresholds are specified in 8-bit resolution, and thus they will be compared with the eight Most Significant Bits of the analog readings to determine when an event should be fired. The given hi-mark must be greater than the lo-mark.

Normally, the "initial state" (high or low) of the analog event definition is determined by the current level of the analog reading at the time the event definition is defined. However, if the current analog reading is between the lo-mark and hi-mark, an initial state cannot be accurately assigned. To deal with this, this method accepts a parameter that defines a default state to be used when the initial state cannot be determined. In all other situations (when the reading at the time of event definition is <= lo-mark or >= hi-mark) this parameter will simply be ignored.

Parameters

- bank - Specifies the event bank (0 or 1).
- channel - Specifies the logical channel (0-7).
- lomark - Low threshold, 8 Most Significant Bits (0-255).
- himark - High threshold, 8 Most Significant Bits (0-255).
- defaultval - If the current reading is between lomark and himark, this specifies whether the event should be considered high or low to begin with. Otherwise, this parameter is ignored. This parameter is a Eth32AnalogEvtDef enumerator type, which has the following valid values:
    - ANEVT_DEFAULT_LOW - Consider the channel to be low

- ANEVT_DEFAULT_HIGH - Consider the channel to be high

Return Value

This method does not return a value.

Remarks

Please note that defining the thresholds with this method does not enable the current connection to actually receive the event notifications when they occur. These must be enabled using the EnableEvent Method. Also note that the analog event thresholds are common to all connections. Changing the thresholds will affect other connections if they are utilizing that particular event.

Because the ETH32 device has two analog event banks, two events can be defined for each logical analog channel on the board. Applications can utilize both event banks independently to implement a number of different event notification schemes.

Example

```
Private Sub example()
    Dim lomark As Long
    Dim himark As Long

    ' Set up error handling for this routine
    On Error GoTo myerror

    Set dev = New Eth32

    ' .... Your code that establishes a connection here

    ' Enable the Analog to Digital Converter
    dev.AnalogState = ADC_ENABLED

    ' Configure logical channel 7 to read the physical channel 7 relative to ground (single-ended)
    ' This is the power-on default anyway, but is shown here for completeness:
    dev.AnalogAssignment(7) = ANALOG_SE7

    ' Configure the analog voltage reference to be the internal 5V source
    dev.AnalogReference = REF_INTERNAL


    ' Define an event that fires when channel 7 goes above 3.5V or
    ' falls below 3.0V.  Remember that the thresholds must be calculated
    ' knowing the voltage reference (in this case 5V).  They also must be
    ' converted to the 8 Most Significant Bits from 10-bit by dividing by 4.
    ' If the current reading happens to be between the low and high threshold,
    ' we will default to the event starting out low.
    lomark = 3# / 5# * 1024 / 4
    himark = 3.5 / 5# * 1024 / 4
    dev.SetAnalogEventDef 0, 7, lomark, himark, ANEVT_DEFAULT_LOW

    ' Now that an event is defined in bank 0, channel 7, enable receiving
    ' events from it.
    ' We'll give this event an arbitrary ID of 8000
    dev.EnableEvent EVENT_ANALOG, 0, 7, 8000

    ' You will now receive events when channel 7 crosses the threshold
    ' to being over 3.5V or crosses to under 3.0V.
```

```
    Exit Sub
myerror:
    MsgBox "ETH32 error: " & dev.ErrorString(Err.Number)
End Sub
```

See Also

EnableEvent Method, GetAnalogEventDef Method, InputAnalog Method

---

SetDirection Method

```
Public Sub SetDirection(ByVal port As Long, ByVal direction As Long)
```

Summary

This method sets the direction register for a digital I/O port, which configures each pin (bit) of the port as an input or output. The direction of each bit of the port can be set individually, meaning that some bits of the port can be inputs at the same time that other bits on the same port are outputs. A 1-bit in the direction register causes the corresponding bit of the port to be put into output mode, while a 0-bit specifies input mode. For example, a value of F0 hex would put bits 0-3 into input mode and bits 4-7 into output mode.

Parameters

- port - The port number (0-5).
- direction - The new direction register for the port.

Return Value

This method does not return a value.

Remarks

Port 3 shares its pins with the analog channels. When the ADC is enabled, all pins of port 3 are forced into input mode. The direction register of port 3 cannot be modified while the ADC is enabled.

The valid range for the direction parameter is any 8-bit number (ranges from 0 to 255). However, note that because ports 4 and 5 are single-bit ports, only bit 0 will have any effect on those ports.

For your convenience, constants for the direction parameter are provided that configure the port bits to be all inputs or all outputs. These are, respectively, DIR_INPUT and DIR_OUTPUT.

Example

```
Private Sub example()

    ' Set up error handling for this routine
    On Error GoTo myerror

    Set dev = New Eth32
```

```
    ' .... Your code that establishes a connection here

        ' Configure all odd bits of port 0 as inputs and even bits as outputs
        '  Direction parameter of 10101010 binary, which is &HAA hex or 170 decimal
        dev.SetDirection 0, 170

    Exit Sub
myerror:
    MsgBox "ETH32 error: " & dev.ErrorString(Err.Number)
End Sub
```

See Also

GetDirection Method, GetDirectionBit Method, SetDirectionBit Method

---

SetDirectionBit Method

```
Public Sub SetDirectionBit(ByVal port As Long, ByVal bit As Long, ByVal direction As Long)
```

Summary

This method alters a single bit of a port's direction register without affecting the value of any other bits. See the SetDirection Method for further description of the direction register.

Parameters

- port - The port number (0-5).
- bit - Which bit within the port to alter (0-7).
- direction - Make the bit an input (0) or an output (1).

Return Value

This method does not return a value.

Remarks

This method alters the specified direction register bit in a single operation directly on the ETH32 device. In other words, it does NOT read the current value over the network, modify it and then write it back. By doing it in a single operation, this avoids the potential of inadvertently overwriting changes made to other bits within the port by other connections.

Port 3 shares its pins with the analog channels. When the ADC is enabled, all pins of port 3 are forced into input mode. The direction register of port 3 cannot be modified while the ADC is enabled.

See Also

GetDirection Method, GetDirectionBit Method, SetDirection Method

---

SetEeprom Method

```
Sub SetEeprom(address As Long, length As Long, buffer() As Byte)
```

Summary

This method stores data into the non-volatile EEPROM memory of the device. Writing to EEPROM memory is a relatively slow process, which will temporarily disrupt event monitoring on the device. See the user manual for specific timing information.

Parameters

- address - The starting location to which data should be stored (0-255).
- length - The number of bytes to store. Valid values for this parameter depend on what is provided for the address parameter. For example, with an address of 0, you may specify a length of all 256 bytes, but with an address of 255, length may only be 1.
- buffer - The data to store into EEPROM memory. This must contain at least as many bytes as you are requesting to store.

Return Value

This method does not return a value.

See Also

GetEeprom Method

SetPwmParameters Method

```
Public Sub SetPwmParameters(ByVal channel As Long, ByVal state As Eth32PwmChannel, _
                            ByVal freq As Single, ByVal duty As Single)
```

Summary

This method is provided for your convenience in working with all of the various PWM settings. It allows you to specify the PWM frequency and the duty cycle of a channel in more familiar terms (hertz and percentage) rather than PWM clock counts. It also puts the appropriate I/O pin into output mode unless you specify that the PWM channel should be disabled. This method internally calls several other API functions to set everything up, therefore replacing calls to PwmBasePeriod Property, PwmDutyPeriod Property, PwmClockState Property, PwmChannel Property, and SetDirectionBit Method with a single call to this method.

Parameters

- channel - Specifies the PWM channel number (0 or 1).
- state - This property is a Eth32PwmChannel enumerator type, which has the following valid values:
  - PWM_CHANNEL_DISABLED - The PWM pin will function as a normal digital I/O pin.
  - PWM_CHANNEL_NORMAL - The PWM pin will function as a PWM output. It will be high for the time specified by the duty period and low for the rest of the PWM base period.
  - PWM_CHANNEL_INVERTED - The PWM pin will function as a PWM output. It will be low for the time specified by the duty period and high for the rest of the PWM base period.
- freq - Specifies the frequency in Hertz. The valid range is 30.5 HZ to 40,000 HZ (40 KHZ)
- duty - Specifies the duty cycle as a percentage (A floating point number from 0.0 to 1.0). This specifies the percentage of each cycle that the channel will be active.

Return Value

This method does not return a value.

Remarks

Note that this method uses the PwmBasePeriod Property to set the PWM base period. Because the PWM base period is shared between both PWM channels, this will affect the other PWM channel if you specify a frequency different than what is already in effect.

Example

```
Private Sub example()

    ' Set up error handling for this routine
    On Error GoTo myerror

    Set dev = New Eth32

    ' .... Your code that establishes a connection here

        ' Set up PWM channel 0 to have a 10 KHZ, 60% PWM signal:
        dev.SetPwmParameters 0, PWM_CHANNEL_NORMAL, 10000, 0.60


    Exit Sub
myerror:
    MsgBox "ETH32 error: " & dev.ErrorString(Err.Number)
End Sub
```

See Also

GetPwmParameters Method

Timeout Property

```
Public Property Timeout As Long
```

Summary

This property configures the internal API timeout on any operations that require a response from the ETH32 device (for example, InputByte). If a method or property routine does not receive a reply from the ETH32 within the timeout period specified, it raises an error with an error number of EthErrorTimeout. This property does not affect the actual ETH32 device, but just the functionality within the API itself. This property does not affect any other Eth32 objects that may be open.

Parameters

This property does not have any parameters.

Value

This property is a Long. It specifies the timeout in milliseconds. A value of zero means that operations should never time out.

VerifyConnection Method

```
Public Sub VerifyConnection()
```

Summary

This method sends a "ping" command (not an ICMP Ping) to the ETH32 device and waits for a response. It can be used to verify that there is still a good connection to the device.

Parameters

This method does not have any parameters.

Return Value

This method does not return a value. If any error occurs, an error will be raised.

See Also

Connect Method, Connected Property, Disconnect Method

Event Handler

When hardware events occur on the ETH32, information about that event is transmitted to your application if you have enabled it using the EnableEvent Method. When this information is received, the Eth32 class notifies your application of the event in a manner that is consistent with the way Visual Basic 6 applications typically process events. That is, the ETH32 events will be processed by your application in a manner very similar to the way that the Click event of a form's command button would be processed.

The event handler function is a function written by you, the programmer. Because it is a function you write, you have complete freedom to inspect whichever aspects of the event data you need to and react however you see fit.

In Visual Basic 6, your event handler function is executed by the same thread as the rest of your code. While this is the same situation as command button Click events or other form events, you should be aware that if some part of your code is tying up the CPU, events will not be processed until that code is done. If you ever need ETH32 events to be processed during a situation such as this, your code may call the CheckEvents Method at the time(s) that any pending events should be processed.

Writing and Registering an Event Handler

Your event handler function's name must start with the name of your object variable, be followed by an underscore and EventFired ('_EventFired'). The event handler must also accept a very specific set of parameters. Fortunately, Visual Basic will automatically create an empty event handler function for you. After you have declared an Eth32 object variable (see the Basic Declaration section above), there will be an entry in the Object drop-down box (the one on the left) at the top of the VB code editor. Simply select that entry and VB will automatically create an empty event handler function. Your code file may now contain something like this:

```
Option Explicit
Dim WithEvents dev As Eth32


Private Sub dev_EventFired(ByVal id As Long, ByVal eventtype As Long, ByVal port As Long, _
                           ByVal bit As Long, ByVal prev_value As Long, ByVal value As Long, _
                           ByVal direction As Long)

End Sub
```

The event handler receives the exact same parameters as the members of the eth32_event structure. Please see description of that structure for an explanation of their meaning. Note that the parameters are passed to the event handler individually instead of as a structure due to restrictions of the VB5/6 language.

Configuration / Detection Functionality

Most of the network configuration and detection functionality of the ETH32 API is contained in the Eth32Config class. If plugins are used to find information about the PC's network interfaces and/or to utilize a sniffer library, that functionality is provided by the Eth32ConfigPlugin class. These classes, their members, and associated structures are described below.

Error Handling

Error codes for the Configuration / Detection Functionality are defined in the EthError enumerator along with the error codes for the main API. Error codes can be translated into a string using the ErrorString Method of the main Eth32 class.

Structures

eth32cfg_ip Structure

The eth32cfg_ip structure holds an IP address in binary form. It is used to represent IP address information in the ETH32 device configuration structure, to specify the broadcast address, and to retrieve IP address information about the PC's network interfaces.

```
Public Type eth32cfg_ip
    buf(3) As Byte
End Type
```

- buf - Array containing individual octets of the IP address. Index 0 contains the most significant, e.g. 192 from the address 192.168.1.100

eth32cfg_mac Structure

The eth32cfg_mac structure holds a MAC address in binary form. It is used within the ETH32 device configuration structure.

```
Public Type eth32cfg_mac
    buf(5) As Byte
End Type
```

- buf - Array containing individual octets of the MAC address. Index 0 contains the first and most significant octet.

eth32cfg_data Structure

The eth32cfg_data structure holds all of the network configuration and device information data for a particular ETH32 device. It is provided to your application when retrieving information about detected devices. Your application can also fill in or modify the information and provide it to the API to store new configuration into a device.

```
Public Type eth32cfg_data
    product_id As Byte
    firmware_major As Byte
    firmware_minor As Byte
    config_enable As Byte
    mac As eth32cfg_mac
    pad1 As Byte
    pad2 As Byte
    serialnum_batch As Integer
    serialnum_unit As Integer
    config_ip As eth32cfg_ip
    config_gateway As eth32cfg_ip
```

```
    config_netmask As eth32cfg_ip
    active_ip As eth32cfg_ip
    active_gateway As eth32cfg_ip
    active_netmask As eth32cfg_ip
    dhcp As Byte
End Type
```

- product_id - Contains the product ID code for the device. This will be 105 for ETH32 devices. This makes up a portion of the serial number.
- firmware_major - Contains the major portion of the firmware version, e.g. 3 from 3.000
- firmware_minor - Contains the minor portion of the firmware version, e.g. 0 from 3.000
- config_enable - Nonzero if the device's Allow Config switch is set to Yes
- mac - The MAC address of the device
- pad1 - Reserved. Padding byte for proper structure alignment
- pad2 - Reserved. Padding byte for proper structure alignment
- serialnum_batch - The batch number portion of the device's serial number
- serialnum_unit - The unit number portion of the device's serial number
- config_ip - The static IP address stored in the device. This is ignored if DHCP is active.
- config_gateway - The static gateway IP address stored in the device. This is ignored if DHCP is active.
- config_netmask - The static network mask stored in the device. This is ignored if DHCP is active.
- active_ip - The IP address being used by the device, whether it was provided by DHCP or statically configured.
- active_gateway - The gateway IP address being used by the device, whether it was provided by DHCP or statically configured.
- active_netmask - The network mask being used by the device, whether it was provided by DHCP or statically configured.
- dhcp - Nonzero if DHCP is being used by the device, or zero if the static settings (config_ip, etc) are being used.

If a device is using DHCP, then active_ip will most likely be different than the static (stored) config_ip, and so on for the gateway and netmask. If DHCP is not being used, then active_ip will be the same as config_ip, and so on for the gateway and netmask.

When using this structure with the SetConfig Method, you may modify the config_ip, config_gateway, config_netmask, and dhcp members in order to update the corresponding settings within the ETH32 device. The other members of the structure should not be modified, since they will either be ignored, or are required for the new configuration to be accepted by the device. Specifically, the MAC address and serial number information must match the device's information, or the device will ignore the new configuration data.

Eth32ConfigPluginInterface Structure

The Eth32ConfigPluginInterface structure holds information about a network interface card of the PC. This information can be provided by a plugin loaded into the ETH32 API.

```
Public Type Eth32ConfigPluginInterface
    Ip As eth32cfg_ip
    Netmask As eth32cfg_ip
    InterfaceType As Eth32ConfigInterfaceType
    StandardName As String
    FriendlyName As String
    Description As String
End Type
```

- Ip - The IP address of the network interface
- Netmask - The network mask of the network interface
- InterfaceType - The type of network that this network interface is for. This can be one of these values:
    - ETH32CFG_IFTYPE_NONE - This is used if the current plugin doesn't provide information about the network interface type.
    - ETH32CFG_IFTYPE_OTHER - This is used if the plugin provides information about the interface type, but it isn't one of the predefined constants.
    - ETH32CFG_IFTYPE_ETHERNET - Ethernet interface
    - ETH32CFG_IFTYPE_TOKENRING - Token Ring interface
    - ETH32CFG_IFTYPE_FDDI - FDDI (Fiber Distributed Data Interface) interface
    - ETH32CFG_IFTYPE_PPP - PPP (Point-to-Point Protocol) interface
    - ETH32CFG_IFTYPE_LOOPBACK - Local loopback interface (e.g. 127.0.0.1)
    - ETH32CFG_IFTYPE_SLIP - SLIP (Serial Line Internet Protocol) interface
- StandardName - This is typically an internal identifier string that identifies the interface, but is not very human-readable. The exact value depends on the plugin being used.

- FriendlyName - The human-readable name for the interface. For example, Local Area Connection. This member will be empty when the WinPcap plugin is being used.
- Description - A description of the interface. The value of this member depends on the plugin being used, but typically includes the manufacturer or model of the card. This member will be available when using the System plugin or when using the WinPcap plugin.

Eth32Config Member Reference

BroadcastAddress Property

```
Friend Property BroadcastAddress As eth32cfg_ip
```

Summary

This read/write property defines the broadcast address that will be used when sending out queries or new configuration data to ETH32 devices. It defaults to 255.255.255.255, which works well in most situations.

Parameters

This property does not have any parameters.

Value

This property is a eth32cfg_ip Structure.

See Also

BroadcastAddressString Property

BroadcastAddressString Property

```
Public Property BroadcastAddressString As String
```

Summary

This read/write property returns or alters the same information as the BroadcastAddress Property, but in string format.

Parameters

This property does not have any parameters.

Value

This property is a string representation of the broadcast address.

See Also

BroadcastAddress Property

DiscoverIp Method

```
Friend Function DiscoverIp(ByVal filter As Eth32ConfigFilter, mac As eth32cfg_mac, _
        ByVal product_id As Byte, ByVal serialnum_batch As Integer, ByVal serialnum_unit As Integer) As Long
```

Summary

This method is used to detect ETH32 devices and their currently-active IP configuration settings. This method allows you to specify a filter so that only the information for the specific ETH32 device that you are interested in will be returned (in case there are multiple ETH32s on the network). This is intended for applications that need to discover the IP of a device that is using DHCP to get its IP address. This method uses a new command to the ETH32 device that is only supported by devices with firmware v3.000 and on. Any older devices on the network will not be detected. The eth32cfg_data structure for devices detected with this method will not have all fields filled in, since the response from the ETH32 does not include all available information. Only the product_id, mac, serialnum_batch, serialnum_unit, active_ip, active_gateway, active_netmask, and dhcp fields will be filled in and valid.

The filter parameter instructs the method which data to filter on. Although this method includes parameters for both MAC and serial number information, they will only be considered if the appropriate flag is present in the filter parameter.

Once this method returns, the configuration data for any devices that have been found will be available through the Result Property.

Parameters

- filter - Specifies which parameters should be considered in discovering the device. If more than one flag is specified, then the device must match BOTH. This parameter is a Eth32ConfigFilter enumerator type, which has the following valid values:
    - ETH32CFG_FILTER_NONE - The parameters will be ignored. All devices will be discovered.
    - ETH32CFG_FILTER_MAC - Only devices matching the provided MAC address will be discovered.
    - ETH32CFG_FILTER_SERIAL - Only devices matching the provided serial number information (id, batch, unit) will be discovered.
- mac - The MAC address of the device you are trying to discover
- product_id - The product ID code (part of the serial number) of the device you are trying to discover. For ETH32 devices, this is 105.

- serialnum_batch - The batch number portion of the serial number for the device you are trying to discover.
- serialnum_unit - The unit number portion of the serial number for the device you are trying to discover.

Return Value

This method returns the number devices that have been found.

Remarks

The number of devices that were found is returned by the method, but also remains available from the NumResults Property. When you are finished with the results, you may free the memory associated with the results using the Free Method. This is done automatically for you if the object is destroyed, or if you call the DiscoverIp Method or the Query Method again on the same object. Note that this means each Eth32Config object holds only one active set of results at one time.

Example

```
Private Sub example()
Dim devdetect As New Eth32Config
Dim dev As New Eth32

' Set up error handling for this routine
On Error GoTo myerror

' Set broadcast address - this line would not
' be necessary since 255.255.255.255 is the default anyway
devdetect.BroadcastAddressString = "255.255.255.255"

' Find a device by serial number -- we can use the ETH32_PRODUCT_ID constant,
' 1 for the batch (AB), and 82 for the unit number.
' This would be serial number 105-AB082 as shown on the device.
Dim tempmac As eth32cfg_mac ' For the MAC address, we just need a variable to pass in.
                            ' It will not be considered since we don't set the flag to use it.
devdetect.DiscoverIp ETH32CFG_FILTER_SERIAL, tempmac, ETH32_PRODUCT_ID, 1, 82

If devdetect.NumResults = 0 Then
    MsgBox "Device not found"
Else
    ' Device was found -- here's a quick example of using the information to now
    ' connect to the device and turn on LED 0.
    dev.Connect devdetect.IpConvertToString(devdetect.Result(0).active_ip)
    dev.Led(0) = True
End If

Exit Sub
myerror:
Dim temp As New Eth32
MsgBox "ETH32 error: " & temp.ErrorString(Err.number)
End Sub
```

See Also

Result Property, NumResults Property, Query Method, Free Method

Free Method

```
Public Sub Free()
```

Summary

This method frees any memory associated with the current set of results held by the object. This can be called after you are finished with the results from the DiscoverIp Method or the Query Method. However, it is called automatically for you when either of those methods is called again, as well as at the time the object is destroyed.

Parameters

This method does not have any parameters.

Return Value

This method does not return a value.

See Also

DiscoverIp Method, Query Method

IpConvert Method

```
Friend Function IpConvert(ByVal ipstring As String) As eth32cfg_ip
```

Summary

This method converts a string representation of an IP address into the eth32cfg_ip binary representation of an IP address. If the string doesn't contain a valid IP address, an EthErrorInvalidIp error will be raised.

Parameters

- ipstring - The IP address to be converted.

Return Value

This method returns an eth32cfg_ip structure with the converted IP address.

See Also

IpConvertToString Method

IpConvertToString Method

```
Friend Function IpConvertToString(ipbinary As eth32cfg_ip) As String
```

Summary

This method converts the eth32cfg_ip binary representation into a string.

Parameters

- ipbinary - The IP address to be converted.

Return Value

This method returns a string representation of the converted IP address.

See Also

IpConvert Method

MacConvert Method

```
Friend Function MacConvert(ByVal macstring As String) As eth32cfg_mac
```

Summary

This method converts a string representation of a MAC address into the eth32cfg_mac binary representation of a MAC address. If the string doesn't contain a valid MAC address, an EthErrorInvalidOther error will be raised.

Parameters

- macstring - The MAC address string to be converted.

Return Value

This method returns an eth32cfg_mac structure with the converted MAC address.

See Also

MacConvertToString Method

MacConvertToString Method

```
Friend Function MacConvertToString(macbinary As eth32cfg_mac) As String
```

Summary

This method converts an eth32cfg_mac binary representation of a MAC address into a string.

Parameters

- macbinary - The MAC address to be converted.

Return Value

This method returns a string representation of the MAC address.

See Also

MacConvert Method

NumResults Property

```
Public Property NumResults As Long
```

Summary

This read-only property indicates how many ETH32 devices were found the last time the DiscoverIp Method or the Query Method was called.

Parameters

This property does not have any parameters.

Value

This property is a Long. The value indicates the number of devices found, and therefore how many items are available through the Result Property.

See Also

Result Property

Query Method

```
Public Function Query() As Long
```

Summary

This method is used to detect all ETH32 devices on the local network segment and all of their available device information and configuration settings. Once this method returns, the configuration data for any devices that have been found will be available through the Result Property.

Parameters

This method does not have any parameters.

Return Value

This method returns the number devices that have been found.

Remarks

The number of devices that were found is returned by the method, but also remains available from the NumResults Property. When you are finished with the results, you may free the memory associated with the results using the Free Method. This is done automatically for you if the object is destroyed, or if you call the DiscoverIp Method or the Query Method again on the same object. Note that this means each Eth32Config object holds only one active set of results at one time.

As opposed to the DiscoverIp Method, which is only supported by devices with firmware 3.000 and greater, the Query Method detects all devices with all firmware versions. This method sends several queries out repeatedly in case any queries or responses are lost on the network. It also delays for a short while to listen for responses. Because of this, the DiscoverIp Method method will be faster if you are looking for a specific device, know its MAC address or serial number, and know it is running firmware v3.000 or greater.

Example

```
Private Sub example()
Dim devdetect As New Eth32Config
Dim i As Long

' Set up error handling for this routine
On Error GoTo myerror

' Set broadcast address - this line would not
' be necessary since 255.255.255.255 is the default anyway
devdetect.BroadcastAddressString = "255.255.255.255"

' Find all devices
devdetect.Query

If devdetect.NumResults = 0 Then
    MsgBox "No devices were found."
Else
    For i = 0 To (devdetect.NumResults - 1)
        MsgBox "Device found with IP address of: " & _
               devdetect.IpConvertToString(devdetect.Result(i).active_ip)
    Next
End If

Exit Sub
myerror:
Dim temp As New Eth32
MsgBox "ETH32 error: " & temp.ErrorString(Err.number)
End Sub
```

See Also

Result Property, NumResults Property, DiscoverIp Method, Free Method

Result Property

```
Friend Property Result(ByVal index As Long) As eth32cfg_data
```

Summary

This property is used to access the device information and configuration data for each device that was found on the last call to the Query Method or the DiscoverIp Method.

Parameters

- index - The index of the result to return.

Value

This property is a eth32cfg_data Structure. It returns the configuration data for the result at the specified index location.

Remarks

The index is zero-based, which means it can range from zero up to one less than the number of available results (as indicated by the NumResults Property).

See Also

eth32cfg_data Structure, NumResults Property, DiscoverIp Method, Query Method

SerialNumString Method

```
Public Function SerialNumString(ByVal product_id As Byte, _
        ByVal serialnum_batch As Integer, ByVal serialnum_unit As Integer) As String
```

Summary

This method takes the numeric components of the ETH32 serial number and formats a serial number string in the same way that it is printed on the ETH32 device enclosure.

Parameters

- product_id - The product ID portion of the serial number
- serialnum_batch - The batch number portion of the serial number
- serialnum_unit - The unit number portion of the serial number

Return Value

This method returns a string representation of the serial number.

See Also

eth32cfg_data Structure

SetConfig Method

```
Friend Sub SetConfig(config_data As eth32cfg_data)
```

Summary

This method is used to store new configuration settings into an ETH32 device. The device's Allow Config switch must be set to Yes, or the new configuration will be rejected.

Parameters

- config_data - The new configuration data and product identification information

Return Value

This method does not return a value. If any error occurs, an error will be raised.

Remarks

The MAC address and serial number information members of the eth32cfg_data Structure identify which device is to be configured. If those members are not set correctly, the device will simply ignore the settings, or worst-case, if they match a different device you were not intending to configure, that device will accept the new configuration. Therefore, in most cases, although it is not required, it is best to take the eth32cfg_data Structure from the Result Property, modify as needed, and then provide that to this method.

Under normal circumstances, the device will accept the configuration and return a confirmation packet, which will cause the method to immediately return without raising any errors. If the device's Allow Config switch is set to No, it will return a rejection packet, which will cause the method to raise the EthErrorConfigReject error. If no response is received from the device, the method will raise the EthErrorConfigNoAck error after a short timeout.

See Also

eth32cfg_data Structure

Eth32ConfigPlugin Member Reference

ChooseInterface Method

```
Public Sub ChooseInterface(ByVal index As Long)
```

Summary

This method selects one of the available network interfaces on the PC as the interface on which the ETH32 Configuration / Detection API (Eth32Config class) should sniff for responses from ETH32 devices. This does not affect the main API functionality (the Eth32 class). The interface list must have been previously obtained using the GetInterfaces Method and the provided index must be a valid index within that list. Currently, this function is only applicable when the WinPcap plugin is loaded. Otherwise, the EthErrorNotSupported error will be raised.

Parameters

- index - The index of the interface in the previously-obtained interface list which should be chosen for sniffing responses

Return Value

This method does not return a value.

See Also

GetInterfaces Method

Free Method

```
Public Sub Free()
```

Summary

This method frees any memory associated with a the network interface list previously obtained using the GetInterfaces Method. This is done automatically if the GetInterfaces Method is called again later, but note that you must call Free on any Eth32ConfigPlugin objects in the same application process (if they have called the GetInterfaces Method) before loading a different plugin with the Load Method.

Parameters

This method does not have any parameters.

Return Value

This method does not return a value.

See Also

GetInterfaces Method

GetInterfaces Method

```
Public Function GetInterfaces() As Long
```

Summary

This method loads the list of available network interface cards on the PC. A plugin which provides this functionality must be loaded first before calling this method. This functionality is provided by both the System and the WinPcap plugins, but not by the None plugin. Once the method returns, details of each interface can be accessed through the NetworkInterface Property

Parameters

This method does not have any parameters.

Return Value

This method returns the number of network interface cards in the list. This number will also remain available from the NumInterfaces Property.

Remarks

If the currently-loaded plugin does not provide this functionality, an EthErrorNotSupported error will be raised.

The memory used by the interface list can be freed with the Free Method. The only time this needs to be done manually is when one plugin (other than None) has been loaded, Eth32ConfigPlugin object(s) with interface list(s) are open, and you are getting ready to load a different plugin with the Load Method. This is due to the fact that the loaded plugin affects the entire process, so it is up to you as the programmer to ensure that any active Eth32ConfigPlugin objects are Free'd before changing the plugin.

See Also

Load Method, NetworkInterface Property, Free Method

Load Method

```
Public Sub Load(ByVal plugin_type As Eth32ConfigPluginType)
```

Summary

This method loads one of the pre-defined plugins. The currently-loaded plugin affects the entire process in terms of the Configuration and Detection functionality (the Eth32Config class), but does not affect the main functionality of the API (the Eth32 class). See the Plugins topic for more information.

Parameters

- plugin_type - The plugin to be loaded. This can be one of the following options:
  - ETH32CFG_PLUG_NONE - No plugin loaded. This is the default if Load is never called. If another plugin is loaded, calling Load with this option will remove the loaded plugin.
  - ETH32CFG_PLUG_SYS - System plugin. The Windows API is used to provide information about the network interfaces on the PC. Using this plugin does not affect how queries are sent out or how responses are received.
  - ETH32CFG_PLUG_PCAP - WinPcap plugin. The WinPcap library is used to provide information about the network interfaces as well as to sniff for ETH32 responses on the chosen interface.

Return Value

This method does not return a value.

Remarks

If a plugin is attempted to be loaded that is not present on the system, an EthErrorNotSupported error will be raised.

When one plugin (other than None) has been loaded and Eth32ConfigPlugin object(s) with interface list(s) are open, you must make sure that the Free Method of each Eth32ConfigPlugin object is called before changing the plugin with this method. This is due to the fact that the loaded plugin affects the entire process, so it is up to you as the programmer to ensure that any active Eth32ConfigPlugin objects are Free'd before changing the plugin.

See Also

Free Method

NetworkInterface Property

```
Friend Property NetworkInterface(ByVal index As Long) As Eth32ConfigPluginInterface
```

Summary

This read-only property provides access to the information about each of the network interfaces in the list, which must be previously obtained by calling the GetInterfaces Method.

Parameters

- index - The index of the interface within the list

Value

This property is a Eth32ConfigPluginInterface Structure. It returns the interface information for the result at the specified index location.

Remarks

The index is zero-based, which means it can range from zero up to one less than the number of available interfaces (as indicated by the NumInterfaces Property).

See Also

GetInterfaces Method, NumInterfaces Property

NumInterfaces Property

```
Public Property NumInterfaces As Long
```

Summary

This read-only property indicates how many network interfaces are in the list that was obtained by calling the GetInterfaces Method and which are now available through the NetworkInterface Property.

Parameters

This property does not have any parameters.

Value

This property is a Long. The value indicates the number of interfaces in the list.

See Also

GetInterfaces Method, NetworkInterface Property

Other Languages

If you are using a language that is not mentioned in this document, you may be able to use the API by making calls directly to the eth32api.dll file if your language supports doing so. Like the Windows API, all functions use Standard Calling Convention (stdcall). All function prototypes and associated structures and constants for the API are listed in the eth32.h header file.

Otherwise, the network protocol used to communicate with the ETH32 device is fully documented in another document (ETH32 Protocol Reference). If for any reason you need to communicate with the device without using the API, please refer to that document.