ZBasic for the ESP8266

Version 1.0.0

Publication History

September 2015 First publication

Disclaimer

Elba Corp. makes no warranty regarding the accuracy of or the fitness for any particular purpose of the information in this document or the techniques described herein. The reader assumes the entire responsibility for the evaluation of and use of the information presented. The Company reserves the right to change the information described herein at any time without notice and does not make any commitment to update the information contained herein. No license to use proprietary information belonging to the Company or other parties is expressed or implied.

Critical Applications Disclaimer

ELBA CORP. PRODUCTS ARE NOT DESIGNED OR INTENDED TO BE FAIL-SAFE OR TO BE USED IN ANY APPLICATION REQUIRING FAIL-SAFE PERFORMANCE, SUCH AS IN LIFE-SUPPORT OR SAFETY DEVICES OR SYSTEMS, CLASS III MEDICAL DEVICES, NUCLEAR FACILITIES, APPLICATIONS RELATED TO THE DEPLOYMENT OF AIRBAGS, OR ANY OTHER APPLICATIONS WHERE DEFECT OR FAILURE COULD LEAD TO DEATH, PERSONAL INJURY OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE (INDIVIDUALLY AND COLLECTIVELY, “CRITICAL APPLICATIONS”). FURTHERMORE, ELBA CORP. PRODUCTS ARE NOT DESIGNED OR INTENDED FOR USE IN ANY APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE OR AIRCRAFT. CUSTOMER AGREES, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE ELBA CORP. PRODUCTS, TO THOROUGHLY TEST THE SAME FOR SAFETY PURPOSES. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, CUSTOMER ASSUMES THE SOLE RISK AND LIABILITY OF ANY USE OF ELBA CORP. PRODUCTS IN CRITICAL APPLICATIONS.

Trademarks

Table of Contents

This page is intentionally blank

ZBasic for the ESP8266

Introduction

The ZBasic Compiler and IDE are capable of producing applications for ESP8266 devices. Because the ESP8266 is different in many respects from the Atmel AVR devices (the original targets of ZBasic) there are necessarily some differences in how ZBasic applications must be structured and in the set of ZBasic System Library routines that are available. There are several new ZBasic System Library routines that are specific to the ESP8266, some routines that that different capabilities and limitations, and some routines that are not available at all for the ESP8266. More details on this topic are found later in this document.

Also, the intrinsic data type `Double` is available for the ESP8266, providing 64-bit floating point in addition to `Single` which provides 32-bit floating point. Both data types may be used in a ZBasic application and automatic conversions are performed when assigning one type to the other (but with loss of precision when `Double` is converted to `Single`). One caveat is that a variable of one of the two floating point data types cannot be passed by reference to a subroutine that is expecting the other data type. The workaround would be to introduce an interim variable of the expected type, assign the value to it and then pass it to the subroutine.

ZBasic for AVR devices is capable of producing either single-task or multi-task applications. For these devices, the ZBasic `Main()` routine is usually structured as some initialization code (which may invoke additional tasks) followed by an infinite loop that performs the intended function of the application. In contrast, a ZBasic application for the ESP8266 is single-task only and consists of an optional initialization subroutine and a `Main()` routine that performs a small amount of work and then returns. The "operating system" of the ESP8266 repeatedly invokes the `Main()` routine in addition to servicing the WiFi interface and performing other system functions. It is recommended that each pass through the `Main()` routine occupy no more than 30 milliseconds or so. Consuming more time that that may interfere with WiFi processing and, further, if too much time is consumed (on the order of seconds) the ESP8266 watchdog timer will trigger a device reset. If an activity to be performed requires more than about 30mS, it can either be broken up into smaller units or the System Library routine `Yield()` may be called occasionally to allow the OS to perform its necessary functions.

With that brief introduction in mind, here is a simple "Hello world" ZBasic application for the ESP8266.

```
Option ConsoleSpeed 230400
Option UserPostInit myPostInit

Const pin as Byte = A.2
Dim state as Integer
Dim pinState as Byte

Sub Main()
   If (state = 0) Then
       pinState = pinState Xor 1
       Call PutPin(pin, pinState)
       Debug.Print "Hello, world!"
   End If
   state = state + 1
   If (state >= 1000) Then
       state = 0
   End If
   Call Sleep(1)
End Sub

Sub myPostInit()
   Call PutPin(pin, pinState)
   Debug.Print
End Sub
```

This simple application outputs the `"Hello, world!"` message on UART0 every second and also toggles the specified pin, in this case A.2 which is GPIO2 on the ESP8266. Notice, particularly, that the `Main()` routine is a straight line of execution that performs some simple operations and then invokes the `Sleep()` routine for 1 millisecond. before returning. The 1 second timing is achieved by counting the number of times that `Main()` is invoked and outputting the message on one in a thousand passes. This simple application could be expanded to perform various other operations on any of the other 999 passes.

Note, also, the use of the compiler directive `Option UserPostInit`, which is specific to the ESP8266, to specify an initialization routine. In this particular application the initialization could just as well have been omitted but it was included to demonstrate how that is done.

Building an application for the ESP8266 using the ZBasic IDE is essentially the same as for any other ZBasic target. The target device is selected using "Device Options" dialog, available via the "Options|Device Options..." menu item. If the ESP8266 target is not already present in the Target Device dropdown, it can be added by clicking the "Edit Target List..." button. In the resulting dialog, two lists are presented: one with devices that are to be shown in the Target Device dropdown (on the previous dialog) and one with devices that are to be omitted. One or more devices can be moved from one list to the other by selecting the device of interest and then clicking one of the "arrow" buttons appearing between the list to effect the move in one direction or the other.

A subsequent section of this document describes the process of downloading the application to the ESP8266.

ESP8266 Pin Mapping

ESP8266 devices are available in many different forms including the ESP-01 and the ESP-12. Each of the available devices has a different set of pins available, different size of Flash memory, etc. ZBasic doesn't have an awareness of the different forms of the ESP8266. Rather, it is up to you as the application programmer to write the application with the capabilities of your target device in mind.

The ESP chip itself has 17 I/O pins denoted as GPIO0 through GPIO16. The I/O pins can be referred to by their ordinal numbers 1 through 17. Additionally, GPIO0 through GPIO15 may be referred to as A.0 through A.15 and GPIO16 may be referred to as B.0. On most ESP8266 boards, GPIO6 through GPIO11 are dedicated for controlling the serial Flash chip and are, therefore, not available to applications. This information, and other details, are summarized in the table below. The shaded rows indicate pins that are generally not available to an ESP8266 application.

ESP8266 Pin Naming and Special Functions

GPIO Pin	ZBasic Name	Special Functions
GPIO0	A.0 or 1	Boot signal
GPIO1	A.1 or 2	UART0 TxD
GPIO2	A.2 or 3	UART1 TxD
GPIO3	A.3 or 4	UART0 RxD
GPIO4	A.4 or 5	
GPIO5	A.5 or 6	
GPIO6	A.6 or 7	Flash control - SCK
GPIO7	A.7 or 8	Flash control - D0
GPIO8	A.8 or 9	Flash control - D1
GPIO9	A.9 or 10	Flash control - D2
GPIO10	A.10 or 11	Flash control - D3
GPIO11	A.11 or 12	Flash control - CS
GPIO12	A.12 or 13	
GPIO13	A.13 or 14	
GPIO14	A.14 or 15	
GPIO15	A.15 or 16	
GPIO16	B.0 or 17	

ZBasic Compiler Directives

Some of the compiler directives available for AVR devices are also available for ESP8166 while others are not applicable to ESP8266 devices. The following list gives the option words (e.g. `Option HeapSize`) that are not allowed or are not useful for the ESP8266.

Arduino	HeapSize
AtnChar	Language
CodeLimit	OCModulateEnable
DefaultISR	RamSize
ExtRamConfig	SerialReadStrobe
HeapLimit	TaskStackMargin
HeapReserve	X10Interrupt

Other than these and those described explicitly below, all other compiler directives are allowed and function as described in the ZBasic Language Reference manual.

The following compiler directives are applicable to the ESP8266 (possibly with some limitations or restrictions as noted). Those that are unique to the ESP8266 are preceded by an asterisk.

*Option UserInit

This directive allows you to specify an initialization routine that is executed early in the startup procedure. The name of the subroutine should be specified following the `UserInit` keyword with at least one intervening space. For reference purposes, the subroutine specified by this compiler directive will be executed after completing low-level ZBasic initialzation, including console serial port initialization, but before the ZBasic sign-on (if enabled). It is important to note that at the time the UserInit subroutine is invoked the ZBasic module-level variables and objects have not yet been initialized. For reference purposes, all of this initialization is done in the ESP8266 `user_init()` routine.

Example: `Option UserInit myInit`

*Option UserPostInit

This directive allows you to specify an initialization routine that is executed after the initialization is complete including the initialization of ZBasic module-level variables and objects. The name of the subroutine should be specified following the `UserPostInit` keyword with at least one intervening space. For reference purposes, this routine is invoked in the "initialization complete" callback set by the ESP8266 `system_init_done_cb()` API.

Example: `Option UserPostInit myPostInit`

*Option PersistentSize

ZBasic reserves a 4K block of Flash memory to serve as the ZBasic Persistent data store. Due to the design of the Flash chip, this entire block is written each time that one or more bytes are changed. Since it is usually the case that the entire block is not changed, it must be updated using a read-modify-write strategy and a block of memory must be temporarily allocated from the heap for this purpose.

Since many ZBasic applications for the ESP8266 will use little or none of the Persistent memory, the default Persistent size is much smaller than the maximum in order to reduce the demand on heap memory for the update operation. The default Persistent size is 128 bytes of which 16 bytes are reserved for "system" use. If your application needs more than the default 112 bytes of Persistent memory, you can specify a larger size using this compiler directive, up to the maximum of 4096. On the other hand, if your application uses fewer than the default 112 bytes you could also specify a smaller size using this

compiler directive in order to reduce the heap use. The smallest PersistentSize that can be specified is 16 which would leave no space for your application's Persistent data.

Example: `Option PersistentSize 512`

Option <pin>

In addition to the four states (`zxInputTriState`, `zxInputPullUp`, `zxOutputLow` and `zxOutputHigh`) two additional state modifiers, `zxOpenDrain` and `zxActivePullup`, may be specified for A.0 through A.15. Active pullup is the default mode for these pins.

Option TargetDevice

The compiler is directed to generate code for the ESP8266 when the target device is specified as `ESP8266`. That said, it is preferable to select the target device in the IDE rather than specifying it in the application source code.

Option ConsoleSpeed

This directive may be used to specify the baud rate to which UART0 is initialized as part of the ZBasic startup. The default console speed is 19200.

Option MainTaskStackSize

The application's `Main()` subroutine is managed internally as if it were a task with its own separate stack. By default, the size of this stack is 4096 bytes. This stack is used for all of the variables allocated locally within ZBasic procedures (excluding those with the `Static` attribute) and for the call/return tracking. This stack use also includes space used by ZBasic System Library routines.

The System Library function `System.TaskHeadroom()` can be used to determine the amount of unused space in the task stack. This can be helpful in adjusting the stack size to meet the needs of your application.

Option RTC

This compiler directive may be used to enable the ZBasic real time clock. If it is enabled, the ESP8266 RTC hardware is used to update the ZBasic RTC tick count. By default, the ZBasic RTC updates the tick count every millisecond but you may specify a different rate using the device parameter `RTCFrequency`.

Option Include

Several pre-defined data types are available for the ESP8266 as noted below. Each of these may be included in your application by giving the name following `Option Include`.

```
Structure WifiScan_t
   Dim ssid as String          ' SSID
   Dim chan as Byte            ' channel number
   Dim auth as Byte            ' authorization mode
   Dim rssi as Integer         ' relative signal strength
   Dim mac(1 to 6) as Byte     ' MAC address
End Structure
```

```
Union IPAddress_t
   Dim w as UnsignedLong
   Dim b(1 to 4) as Byte
End Union

Structure Microtime_t
   Dim timerTicks as UnsignedInteger
   Dim fastTicks as UnsignedLong
End Structure
```

Option DeviceParameter

The operating frequency of the device may be specified using the device parameter `ClockFrequency`. The allowable values are 80MHz (the default) or 160MHz using the designators `"80M"` and `"160M"` or the numeric equivalents 80000000 and 160000000.

Several new device parameters apply solely to the ESP8266.

`FlashSize`	This device parameter may be used to specify the size of the serial Flash chip using the designators `"512K"` (the default), `"1M"`, `"2M"`, `"4M"`, `"8M"` or their numeric equivalents 524288, 1048576, 2097251, 4194304, and 83886808, respectively.
`FlashFrequency`	This device parameter may be used to specify the operating frequency of the serial Flash chip using the designators `"20M"`, `"26M"`, `"40M"` (the default) `"80M"`, or their numeric equivalents 20000000, 26000000, 40000000 and 80000000, respectively.
`FlashMode`	This device parameter may be used to specify the operating mode of the Flash chip using the designators `"QIO"` (the default), `"QOUT"`, `"DIO"` or `"DOUT"`.

It is important to note that specifying the wrong operating mode or frequency for the Flash chip may result in your application not working as may specifying the Flash size larger that it actually is. Specifying the Flash size smaller than it actually is has no ill effect other than to result in less Flash memory being available to the application.

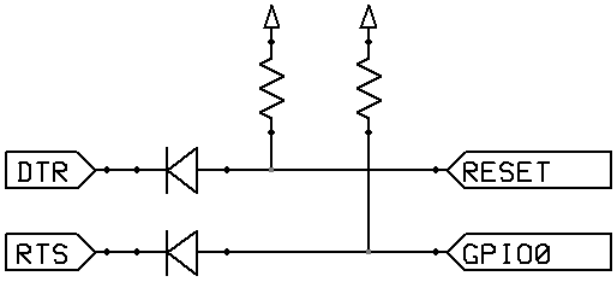
Downloading ESP8266 Applications to the Device

All ESP8266 devices have a built-in bootloader. In order to get the device to run the bootloader (as opposed to the application), the protocol is to hold GPIO0 low while resetting the device. On all ESP8266 devices, this process can be performed manually using a switch or jumper and this reset method is, therefore, referred to as the Manual Reset method.

Some devices have special circuitry that allows the serial port DTR and/or RTS signals to be used to get the device into the bootloader. There are several known methods available on different boards but the effect of all of the is the same, i.e. to hold GPIO0 low while resetting the device. The following text describes the known methods and provides a name for each of them. Although pullup resistors are shown on the RESET and GPIO0 lines in the diagrams below, it is important to note that any particular board may have these present already.

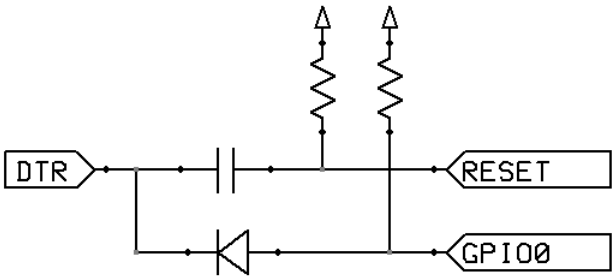
Auto Reset Mode

In this reset mode, the DTR signal is connected to the RESET pin (either directly or via a diode) while the RTS signal is connected to GPIO0 (again, either directly ore via a diode). An example circuit for this mode is shown below.



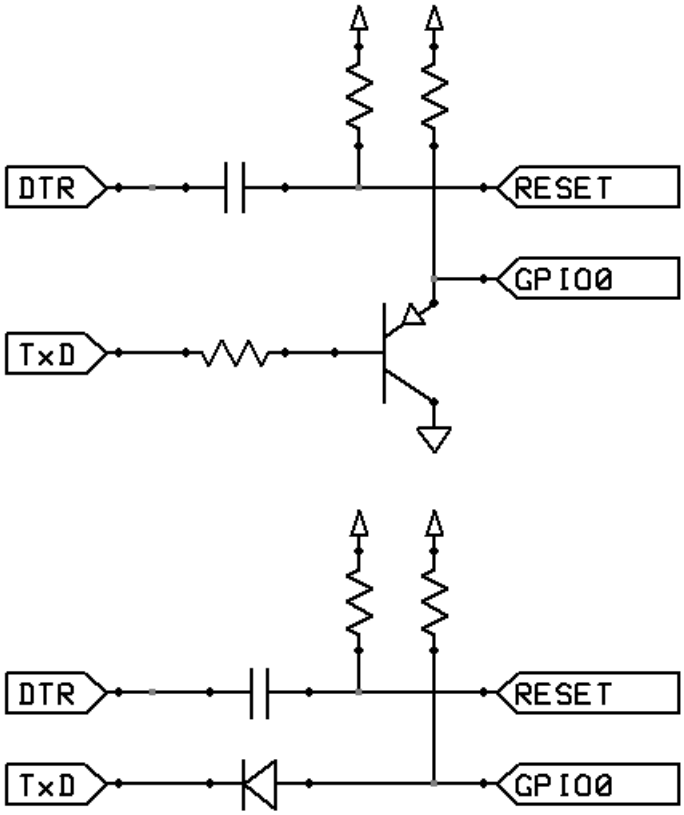
DTR Only Reset Mode

In this reset mode, the DTR signal is connected to the RESET pin via a diode and also connected to GPIO0 (either directly ore via a diode). This is useful with some types of USB-Serial adapters that do not break out the RTS signal. An example circuit for this mode is shown below.



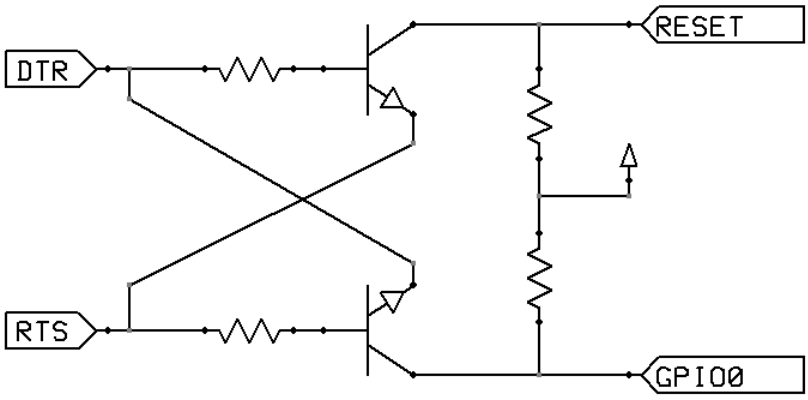
Wifio Reset Mode

This is another reset mode that is useful when RTS is not available. Here, the DTR signal is connected to RESET via a capacitor and GPIO0 is held low via a transistor by transmitting a serial break, effectively holding the TxD line (the RxD input to the ESP8266) low for a period of time, typically about 250mS. The simpler alternative shown second below seems to work just as well.



NodeMCU Reset Mode

In this reset mode, implemented on the NodeMCU Development board, both DTR and RTS are used but they connect to RESET and GPIO0 through a pair of cross-connected NPN transistors. This allows either RESET or GPIO0 to be pulled low but only when DTR and RTS are in opposite logic states.



When the ESP8266 is the selected target device, the Serial Options dialog (available via Options|Serial Options...) will contain a set of radio buttons in the lower righthand group box allowing the choice of reset modes. The captions on the radio buttons are similar to the names used in the descriptions above. Once the reset mode has been selected, pressing F5 with your project loaded wil cause the IDE to attempt to connect to the device and start the download process. If the manual reset mode is selected, you'll have to ground the GPIO0 pin and reset the device manually while the IDE is attempting to establish communications - the window for effecting the reset is about 5 seconds.

It should be noted that the desired download speed may be selected on the Serial Options dialog. The ESP8266 device has "auto-baud" code so there isn't anything that needs to be done on the device itself

when the download baud rate is changed. Users have reported success with baud rates as high as 460800 but higher rates may also work if your serial hardware support them.

Pins Supported for I/O Routines

Because GPIO16 (B.0) is realized using different circuitry than is used for GPIO0-GPIO15 (A.0-A.15), not all of the I/O routines support GPIO16. The I/O routines that do support GPIO16 are listed below.

```
GetPin()          PutPin()
PulseIn()         PulseOut()
```

Other differences between the port A pins and the port B pin are that B.0 cannot generate a pin change event nor it is able to operate in "open drain" mode. Additionally, B.0 has no "input with pullup" mode.

Differences in System Library Routines

Some of the ZBasic System Library routines operate with different timing and/or different restrictions on the ESP8366 as compared to AVR targets. The discussion below summarizes the differences.

CountTransitions() - The sampling loop is about 65 CPU cycles (800nS at 80MHz). Although interrupts are disabled, non-maskable interrupts may still occur that cause a pair of closely spaced transitions to be missed.

Delay() - The resolution of the delay time is 1mS.

GetElapsedMicroTime()
GetMicroTime() - The Microtime_t structure is 6 bytes in length. The resolution of the 32-bit *fastTicks* element is 1mS while the resolution of the 16-bit *timerTicks* element is 200nS.

OpenI2C() - Only one I2C channel is supported and, since the ESP8266 has no I2C hardware, the I2C protocol is implemented in software using the specified pins for SDA and SCL. The default bit rate is about 450KHz. The bitRate value of 10 will yield a bit rate of about 400KHz, 50 will yield about 200KHz and 120 will yield about 100KHz.

OpenPWM() - PWM is available only on PortA pins and is implemented in software based on the ESP8266 RTC Timer1 (a 24-bit down counter). Only one PWM signal can be generated at any one time and the channel parameter is the desired pin number. The minimum supported base frequency is 1.0Hz and the maximum is about 78KHz. Note, however, that lower frequencies are to be preferred due to the lower load imposed on the CPU.

OpenSPI() - Only one SPI channel is supported and the SPI protocol is implemented in software using the pins specified in the `DefineSPI()` call. The bit rate portion of the *flags* parameter is ignored. The SPI clock frequency is approximately 1MHz when running at 80MHz.

PulseIn() - The value returned is the number of RTC Timer2 ticks, divided by the Register.TimerSpeed1 divisor value, corresponding to the pulse width with a resolution of 200nS. Although interrupts are disabled, non-maskable interrupts may still occur that can cause a transition to be detected later than it occurs.

PulseOut() - The units of the pulse width is 200nS multiplied by the Register.TimerSpeed1 divisor. Although interrupts are disabled, non-maskable interrupts may still occur that cause a transition to be later than it should be.

PutPin() - When generating a pulse (*mode* = 5), the pulse width is about 250 nS. Also, an additional mode OpenDrain (mode=9, zxOpenDrain) is available. The opposite mode is ActivePullup (mode=10, zbActivePullup).

ShiftInEx() - With no speed control the clock period is approximately 1.5uS at 80MHz. With speed control, the extra delay per bit is a number of CPU cycles approximately equal to 5 times the *bitTime* parameter value. When running at 160MHz, the *bitTime* parameter value is doubled in order to maintain the same approximate timing.

ShiftOutEx() - With no speed control the clock period is approximately 950 nS at 80MHz. With speed control, the extra delay per bit is a number of CPU cycles approximately equal to 5 times the *bitTime* parameter value. When running at 160MHz, the *bitTime* parameter value is doubled in order to maintain the same approximate timing.

Sleep() - The resolution of the sleep time is 1mS.

System Library Routines Not Available for the ESP8266

The list below gives the System Library Routines that are not supported on the EPS8266. The reason that they are not supported is, in most cases, because the ESP8266 lacks the hardware necessary to support them or that it was deemed impractical to support them. The boldface entries are not supported because they relate to multi-tasking. A few of the routines are not supported because they were thought not to have high enough utility value in the ESP8266 environment.

ADCtoCom1	**ExitTask**	PlaySound	**StartTask**
BusRead	FreqOut	ProgMemFind	StatusX10
BusWrite	GetEEPROM	PutDAC	System.HeapHeadRoom
CloseDAC	InputCapture	PutEEPROM	System.HeapLimit
ClosePWM8	InputCaptureEx	PutProgMem	System.HeapSize
CloseX10	**LockTask**	PWM8	**TaskIsLocked**
Com1toDAC	MemAddress	RCtime	**TaskIsValid**
ControlCom	OpenDAC	**ResumeTask**	**UnlockTask**
CPUsleep	OpenI2CSlave	**RunTask**	WaitForInterrupt
DAC	OpenPWM8	SerialGetByte	**WaitForInterval**
DACpin	OpenX10	SerialIn	X10cmd
DefineBus	OutputCapture	SetBoot	ZXCmdMode
DefineCom3	OutputCaptureEx	SetInterval	
DefineX10	PersistentPeek	SetQueueX10	
DelayUntilClockTick	PersistentPoke	StackCheck	

New Subroutines and Functions for the ESP8266

In the descriptions that follow, the parameter types that are accepted by each routine are described. Some parameters accept a specific fundamental data type while others may accept a few similar types. Others accept virtually any parameter type. In order to more succinctly describe the types of parameters accepted, some descriptive type categories are used. For example, the category *integral* is used to connote those types that have the integral characteristic, such as `Byte`, `Integer`, `UnsignedInteger`, `Long` and `UnsignedLong`. The table below indicates which types belong to which categories.

Type Category Membership

Type/Category	any type	numeric	integral	real	signed	int8/16	int16	int32	any 32-bit
`Boolean`	x								
`Bit`	x	x	x			x			
`Nibble`	x	x	x			x			
`Byte`	x	x	x			x			
`Integer`	x	x	x		x	x	x		
`UnsignedInteger`	x	x	x			x	x		
`Long`	x	x	x		x			x	x
`UnsignedLong`	x	x	x					x	x
`Single`	x	x		x	x				x
`Double`	x	x		x	x				
`Enum`	x								
`String`	x								

The remainder of this document presents complete descriptions of each of the System Library routines that is unique to the ESP8266, arranged in alphabetical order.

DeepSleep

Type Subroutine

Invocation DeepSleep(duration, startup)

Parameter	Method	Type	Description
duration	ByVal	integral	The number of microseconds to sleep.
startup	ByRef	anyType	The startup mode upon reawakening.

Discussion

This subroutine may be used to put the ESP8266 in deep sleep mode, substantially reducing power consumption. Only the on-board RTC is kept running while in deep sleep mode.

Startup Mode

Value	Meaning
0	Radio calibration after deep sleep is as specified in byte 108 of esp_init_data_default.bin.
1	Recalibrate the radio after wakeup (increases current consumption.
2	Do not recalibrate the radio after wakeup.
4	Disable the radio after wakeup.

Note that when the specified time expires, the GPIO16 pin will be pulled low. This should be connected to the RESET pin (usually via a diode) in order to apply a reset signal to wake up the device. Because the on-board RTC is kept running during deep sleep, you may store some parameter values (i.e. state information) in the memory area of the on-board RTC to be used durint the post-sleep startup (see RTC.MemRead and RTC.MemWrite). The value of `Register.ResetFlags()` may be used to determine the cause of the reset and therefore control what is done during startup.

Register.ResetFlags Values

Value	Meaning
0	Power-on reset.
1	Hardware watchdog reset.
2	Exception reset.
3	Software watchdog reset.
4	Software reset.
5	Deep sleep awakening reset.
6	External reset.

File.Close

Type Subroutine

Invocation File.Close(handle)

Parameter	Method	Type	Description
handle	ByVal	integral	The handle of the SPIFFS file to be closed, previously returned by `File.Open()`.

Discussion

This subroutine closes the file handle specified. It should be called when operations on the file are completed.

File.Delete

Type Function returning Integer

Invocation File.Delete(filename)

Parameter	Method	Type	Description
filename	ByVal	String	The name of the SPIFFS file to be deleted.

Discussion

This function deletes the named file (if it exists). The return value will be zero if successful and non-zero in case of an error. Note that the file to be deleted should not still be open (see File.Close).

File.Error

Type Function returning Long

Invocation File.Error()

Discussion

After some other SPIFFS function is called that returns an error indication, calling this function will return the specific error code indicating the nature of the problem. A list of the error codes can be found in the SPIFFS source code file "spiffs.h".

File.Mount

Type Function returning Integer

Invocation File.Mount(startAddr, size)

Parameter	Method	Type	Description
startAddr	ByVal	integral	The Flash address for the start of the file system.
size	ByVal	integral	The size, in bytes, of the file system.

Discussion

Before any SPIFFS operations can be performed, the file system must be mounted, giving the address and size of a portion of the Flash chip that is otherwise unused. The address must be on a 1024 byte block boundary and the size must be an integral multiple of 1024. The area to be used for the file system must either have all its bytes set to &Hff (erased) or it must be loaded with a SPIFFS image prepared, for example, with the spiffy.exe utility. The esp_tool.exe utility can be used to erase a region and to flash it with a SPIFFS image.

File.Open

Type Function returning Integer

Invocation File.Open(filename, mode)

Parameter	Method	Type	Description
filename	ByVal	String	The name of the file to be opened.
mode	ByVal	anyIntegral	The mode in which the file will be opened (see discussion).

Discussion

This function opens the named file (subject to the mode value) and returns a non-negative file handle (to be used in other operations) if successful. If the file cannot be opened as specified the return value will be negative.

The *mode* parameter can be a combination of the values in the table below. Note, however, that some combinations are contradictory and will result in failure to open.

Mode Values for File.Open

Value	Value	Meaning
&H01	1	Open an existing file, writes add to the end of the file.
&H02	2	Open an existing file, truncate it to zero length.
&H04	4	Create a new file, deleting an existing file with the same name.
&H08	8	Prepare for read-only operation (writes will fail).
&H10	16	Prepare for write-only operation (reads will fail).
&H18	24	Prepare for read and write operations.
&H20	32	Do not cache write data.
&H44	68	Create a new file, fail if file exists already.

File.Read

Type Function returning Long

Invocation File.Read(handle, destination, count)

Parameter	Method	Type	Description
handle	ByVal	integral	The handle of the file to read, previously returned by `File.Open()`.
destination	ByRef	anyType	Where to place the data read.
arg	ByVal	Single	The number of bytes to read.

Discussion

This function attempts to read the specified number of bytes from the file handle. If successful, the return value will be greater than zero indicating the number of bytes placed in the destination buffer. If fewer bytes than requested were available, the return value will be the smaller number, possibly zero. A negative return value indicates an error occurred.

File.Rename

Type Function returning Integer

Invocation File.Rename(oldName, newName)

Parameter	Method	Type	Description
oldName	ByVal	String	The file to be renamed.
newName	ByVal	String	The new filename.

Discussion

This function attempts to rename the indicated file. If successful, the return value will be zero otherwise the return value will be negative indicating an error.

File.Seek

Type Function returning Long

Invocation File.Seek(handle, offset, origin)

Parameter	Method	Type	Description
handle	ByVal	integral	The handle of the file to read, previously returned by `File.Open()`.
offset	ByVal	integral	The signedoffset to which to seek, relative to the origin parameter.
origin	ByVal	integral	The origin for the seek (see discussion).

Discussion

This function repositions the file in preparation for subsequent read/write operations. The *offset* parameter is treated as a signed value allowing seeks to positions before or after the origin position which is specified as one of the values in the table below.

Origin Designators for Seek Operation

Value	Meaning
`0`	The new position is relative the the beginning of the file.
`1`	The new position is relative to the current position.
`2`	The new position is relative to the end of the file.

Note that negative offsets don't make sense when the origin is the beginning of the file nor do offsets greater than zero when the origin is the end of the file. The value returned is negative if an error occurs, otherwise, the value returned is the new file position. Invoking `File.Seek` with a position of zero and an origin value of 1 results in the current position being returned. Invoking `File.Seek` with a position of zero and an origin value of 2 returns the current file size (but see `File.Size`).

File.Size

Type Function returning Long

Invocation File.Size(handle)

Parameter	Method	Type	Description
handle	ByVal	integral	The handle of the file to read, previously returned by `File.Open()`.

Discussion

The value returned by this function is negative if an error occurs. Otherwise, it is the current size of the file, in bytes.

File.Write

Type Function returning Long

Invocation File.Write(handle, source, count)

Parameter	Method	Type	Description
handle	ByVal	integral	The handle of the file to read, previously returned by `File.Open()`.
source	ByRef	anyType	The data to be written.
arg	ByVal	Single	The number of bytes to write.

Discussion

This function attempts to write a number of bytes of data to the specified file. If an error occurs, the return value will be negative. Otherwise, the return value is the number of bytes written.

Flash.Erase

Type Subroutine

Invocation Flash.Erase(addr, size)

Parameter	Method	Type	Description
addr	ByVal	integral	The start of the Flash region to erase.
size	ByRef	integral	The number of bytes to erase.

Discussion

This subroutine will erase a region of Flash memory. The actual start address for the region is that specified by the *addr* parameter rounded down to a 4K block boundary and the size of the region to be erased will be the specified *size* rounded up to a multiple of 4K bytes.

This routine can be used to erase any part of Flash memory, even that containing parts of the application so it must be used with great care.

Flash.Read

Type Function returning UnsignedInteger

Invocation Flash.Read(addr, dest, count)

Parameter	Method	Type	Description
addr	ByVal	anyIntegral	The Flash address at to begin reading.
dest	ByRef	anyType	Where to place the data read.
count	ByVal	anyIntegral	The number of bytes to read.

Discussion

This function reads an arbitrary region of Flash memory, placing the data at the indicated destination. The return value represents the number of bytes written to the destination.

Flash.Write

Type Function returning UnsignedInteger

Invocation Flash.Write(addr, source, count)

Parameter	Method	Type	Description
addr	ByVal	anyIntegral	The Flash address at to begin reading.
source	ByRef	anyType	The data to be written.
count	ByVal	anyIntegral	The number of bytes to write.

Discussion

This function writes data to Flash memory. It must be used with great care because it can cause parts of the application to be overwritten. The value returned is the number of bytes written.

Net.Address

Type Function returning UnsignedLong

Invocation Net.Address(oct1, oct2, oct3, oct4)

Parameter	Method	Type	Description
oct1	ByVal	Byte	The first octet of the IP address.
oct2	ByVal	Byte	The second octet of the IP address.
oct3	ByVal	Byte	The third octet of the IP address.
oct4	ByVal	Byte	The fourth octet of the IP address.

Discussion

This function returns a 32-bit IPV4 address containing the four octets. For example, to get the IP address for 192.168.0.1 invoke it as `Net.Address(192, 168, 0, 1)`.

Net.Close

Type Function returning Integer

Invocation Net.Close(handle)

Parameter	Method	Type	Description
handle	ByVal	anyIntegral	The handle, previously returned by Net.Open(), to close.

Discussion

This function closes a network handle, making the specified handle available for future operations. The return value will be zero if successful, negative otherwise.

Net.Connect

Type Function returning Integer

Invocation Net.Connect(handle, ipAddr, port, protocol)

Parameter	Method	Type	Description
handle	ByVal	anyIntegral	The handle, previously returned by Net.Open(), to use.
ipAddr	ByVal	UnsignedLong	The IP address of the remote host.
port	ByVal	anyIntegral	The port number to use.
protocol	ByVal	anyIntegral	The protocol indicator.

Discussion

This function attempts to establish a connection to a remote host on the given port number. The *protocol* parameter indicates the desired protocol according to the table below.

Protocol Designators

Value	Meaning
0	Use the TCP protocol.
1	Use the UDP protocol.

The return value will be zero if successful, negative otherwise.

Net.Disconnect

Type Function returning Integer

Invocation Net.Disconnect(handle)

Parameter	Method	Type	Description
handle	ByVal	anyIntegral	The handle, previously returned by Net.Open(), to use.

Discussion

This breaks a connection established by Net.Connect(). The return value will be zero if successful, non-zero otherwise.

Net.GetHostByName

Type Function returning UnsignedLong

Invocation Net.GetHostByName(hostName)

Parameter	Method	Type	Description
hostName	ByVal	String	The host name to lookup.

Discussion

This function returns the IP address of the named host (if found) or the value zero (i.e. 0.0.0.0) if not found.

Net.Listen

Type Function returning Integer

Invocation Net.Listen(handle, port, protocol)

Parameter	Method	Type	Description
handle	ByVal	anyIntegral	The handle, previously returned by Net.Open(), to use.
port	ByVal	anyIntegral	The port number to use.
protocol	ByVal	anyIntegral	The protocol indicator.

Discussion

This function causes the device to begin waiting for a connection request on the indicated *port* number for the given *protocol* (see the table below).

Protocol Designators

Value	Meaning
0	Use the TCP protocol.
1	Use the UDP protocol.

The return value will be zero if successful, negative otherwise. If a callback has been associated with the network handle, the callback will be invoked when a remote host establishes a connection.

Net.Open

Type Function returning Integer

Invocation Net.Open()

Discussion

This function attempts to find an unused network handle. If successful, the non-negative handle will be returned to be used in subsequent network operations. A negative value indicates failure.

Net.RemoteHost

Type Function returning Integer

Invocation Net.RemoteHost(handle, ipAddr, port)
Net.RemoteHost(handle, ipAddr)

Parameter	Method	Type	Description
handle	ByVal	anyIntegral	The handle, previously returned by Net.Open(), to use.
port	ByRef	UnsignedLong	The IP address of the remote host.
protocol	ByRef	UnsignedInteger	The remote port.

Discussion

This function determine the IP address and, optionally, the port of the remote host that is party to an existing connection. The return value will be zero if the parameters of the remote host were successfully determine and a negative value otherwise. This function is most useful when the ESP8266 is listening for a connection. The returned IP address can then be used to control the response.

Net.SendData

Type Function returning Integer

Invocation Net.SendData(handle, data, count)

Parameter	Method	Type	Description
handle	ByVal	anyIntegral	The handle, previously returned by Net.Open(), to use.
data	ByRef	anyType	The data to send.
count	ByVal	anyIntegral	The number of bytes to send.

Discussion

This function attempts to send a block of data to the remote host. If successful, the return value is zero otherwise it is negative. If a callback procedure has been established, it will be invoked when the data is fully transmitted.

Net.SendProgData

Type Function returning Integer

Invocation Net.SendProgData(handle, dataAddr, count)

Parameter	Method	Type	Description
handle	ByVal	anyIntegral	The handle, previously returned by Net.Open(), to use.
dataAddr	ByVal	UnsignedLong	The address of the data to send.
count	ByVal	anyIntegral	The number of bytes to send.

Discussion

Function attempts to send a block of ProgMem data to the remote host. If successful, the return value is zero otherwise it is negative. If a callback procedure has been established, it will be invoked when the data is fully transmitted.

Net.SendStr

Type Function returning Integer

Invocation Net.SendStr(handle, str)

Parameter	Method	Type	Description
handle	ByVal	anyIntegral	The handle, previously returned by Net.Open(), to use.
str	ByVal	String	The string to send.

Discussion

Function attempts to send the characters of a string to the remote host. If successful, the return value is zero otherwise it is negative. If a callback procedure has been established, it will be invoked when the data is fully transmitted.

Net.SetCallback

Type Function returning Integer

Invocation Net.SendStr(handle, callback)

Parameter	Method	Type	Description
handle	ByVal	anyIntegral	The handle, previously returned by Net.Open(), to use.
callback	ByVal	UnsignedLong	The address of the callback procedure.

Discussion

This function associates a callback routine with the network handle. The callback will be invoked for each of several different events related to the network handle. The callback routine must be defined thus:

```
Sub netCallback(ByVal cbtype as Byte, ByVal handle as Integer, _
    ByRef data() as Byte, ByVal cnt as UnsignedInteger)
End Sub
```

In all cases, the *handle* parameter gives the network handle associated with the event. The *cbtype* parameter indicates the type of event that evoked the callback as described in the table below. The *data* and *cnt* parameters are valid only for certain callback types.

Callback Type Designators

Value	Meaning
0	A connection occurred.
1	A disconnect occurred.
2	Data was received, the *data* and *cnt* parameters are valid.
3	A send data function was instigated.
4	The send data request was completed.
5	A reconnection occurred.

Net.SetTimeout

Type Subroutine

Invocation Net.SetTimeout(handle, timeout)

Parameter	Method	Type	Description
handle	ByVal	anyIntegral	The handle, previously returned by Net.Open(), to use.
timeout	ByVal	anyIntegral	The timeout threshold for closing a connection.

Discussion

This subroutine sets the timeout, in seconds. If no activity occurs within the timeout period a connection will be broken. If a callback has been associated with the network handle, the callback will be invoked with the callback type set to 1.

Net.Status

Type Function returning Integer

Invocation Net.Status(handle)

Parameter	Method	Type	Description
handle	ByVal	anyIntegral	The handle, previously returned by Net.Open(), to use.

Discussion

This function returns the status of the network handle. The return value will contain zero or more of the flag values in the table below.

Network Handle Status Bits

Value	Meaning
&H01	The handle is valid.
&H02	The handle is open.
&H04	The handle is connected to a remote host.
&H08	In the connection, the ESP8266 is the client.
&H10	The connection has data not yet sent.

PinChange.Handler

Type Subroutine

Invocation PinChange.Handler(subAddr)

Parameter	Method	Type	Description
subAddr	ByVal	UnsignedLong	The address of a subroutine to invoike on pin change.

Discussion

This subroutine sets a callback procedure to be invoked when an input pin changes state. The callback procedure must be defined as shown by example:

```
Sub pinChangeCB(ByVal flags as UnsignedInteger)
End Sub
```

When the callback is invoked, the bits of the *flags* parameter will have a 1 for each pin A.15 to A.0 that has changed state.

PinChange.Mode

Type Subroutine

Invocation PinChange.Mode(pin, mode)

Parameter	Method	Type	Description
pin	ByVal	Byte	The pin to be enabled/disabled for pin change detection.
mode	ByVal	Byte	The sensitivity mode for the pin (see discussion).

Discussion

This routine enables or disables sensitivity to state change of an input pin (A.0 through A.15 only). The *mode* parameter should have one of the values in the table below.

Pin Change Interrupt Mode

Value	Meaning
0	Disable pin change sensitivity.
1	Invoke callback on a rising edge.
2	Invoke callback on a falling edge.
3	Invoke callback on a both edges.

RTC.MemRead

Type Subroutine

Invocation RTC.MemRead(addr, dest, count)

Parameter	Method	Type	Description
addr	ByVal	anyIntegral	The address at which to begin reading.
dest	ByRef	anyType	Where to put the data read.
count	ByVal	anyIntegral	The number of bytes to read.

Discussion

This on-board RTC of the ESP8266 has a block of RAM, 512 bytes of which are reserved for use by application programs. Although the RAM can be used for any purpose, the fact that the RTC remains under power during deep sleep means that the RAM can be used to store parameter values across a deep sleep cycle.

The zero-based *addr* parameter gives the beginning address, relative to the start of the block reserved for application data.

RTC.MemWrite

Type Subroutine

Invocation RTC.MemWrite(addr, source, count)

Parameter	Method	Type	Description
addr	ByVal	anyIntegral	The address at which to begin writing.
source	ByRef	anyType	The source of the data to write.
count	ByVal	anyIntegral	The number of bytes to write.

Discussion

This on-board RTC of the ESP8266 has a block of RAM, 512 bytes of which are reserved for use by application programs. Although the RAM can be used for any purpose, the fact that the RTC remains under power during deep sleep means that the RAM can be used to store parameter values across a deep sleep cycle.

The zero-based *addr* parameter gives the beginning address, relative to the start of the block reserved for application data.

Wifi.Connect

Type Subroutine

Invocation Wifi.Connect()

Discussion

Invoking this subroutine causes the ESP8266 to attempt to make a connection to the last-specified SSID. In most cases, it is not necessary to use this subroutine because the ESP8266 will automatically attempt to make the WiFi connection each time it boots up.

One could use this subroutine in conjunction with `Wifi.Disconnect()` and `Wifi.SetConfig()` to dynamically change the access point during executions.

Wifi.Disconnect

Type Subroutine

Invocation Wifi.Disconnect()

Discussion

Invoking this subroutine causes the current wireless connection, if any, to be terminated.

Wifi.GetConfig

Type Subroutine

Invocation
Wifi.GetConfig(ssid)
Wifi.GetConfig(ssid, passwd)
Wifi.GetConfig(ssid, passwd, macAddr)

Parameter	Method	Type	Description
ssid	ByRef	String	Where to place the SSID.
passwd	ByRef	String	Where to place the access point password.
macAddr	ByRef	Byte	Where to place the MAC address.

Discussion

This subroutine retrieves the current set of WiFi connection parameters, previously set by `Wifi.SetConfig()`. If the *macAddr* parameter is given, it must refer to a `Byte` buffer that is at least six bytes in length so that it can receive the six bytes of the MAC address.

Wifi.GetHostname

Type Function returning String

Invocation Wifi.GetHostname()

Discussion

This function returns the name identifying the ESP8266 network interface on the network. By default, the host name is derived from the MAC address of the WiFi network interface but it can be set to another name using `Wifi.SetHostname()`. The actual host name is not important for most network operations but having it set to a familiar or recognizable name may facilitate better understanding of DHCP logs, etc.

Wifi.GetIP

Type Function returning UnsignedLong

Invocation Wifi.GetIP(selector)

Parameter	Method	Type	Description
selector	ByVal	anyIntegral	This parameter selects the element of interest (see discussion).

Discussion

The ESP8266 WiFi module contains two network interfaces, one for operating in "station" mode and one for operating in "access point" mode. This function allows retrieval of network parameters for each of the interfaces using several different selector values as shown in the table below.

IP Selector Values

Value	Meaning
0	Return the station IP address.
1	Return the station gateway IP address.
2	Return the station network address mask.
4	Return the AccessPoint IP address.
5	Return the AccessPoint gateway IP address.
6	Return the AccessPoint network address mask.

Wifi.GetMode

Type Function returning Byte

Invocation Wifi.GetMode()

Discussion

This function returns the current Wi-Fi operating mode as previously set by `Wifi.SetMode()`. See the discussion of `Wifi.SetMode()` for more information.

Wifi.Scan

Type Function returning UnsignedInteger

Invocation Wifi.Scan(nodeList, count)

Parameter	Method	Type	Description
nodeList	ByRef	array of WifiScan_t	The array in which to return information about available access points.
count	ByVal	anyIntegral	The number of elements in the *nodeList* array.

Discussion

This function invokes a scan of available Wi-Fi access points, returning the information in the passed array (up to its maximum number of elements). The return value indicates how many elements of the passed array were populated. Note that the array is populated with information on the available access points in the order in which they were detected, without regard to signal strength or any other attribute of the access point.

Each element of the scan node array has the pre-defined structure as shown below.

```
Structure WifiScan_t
   Dim ssid as String          ' SSID
   Dim chan as Byte            ' channel number
   Dim auth as Byte            ' authorization mode
   Dim rssi as Integer         ' relative signal strength
   Dim mac(1 to 6) as Byte     ' MAC address
End Structure
```

This pre-defined structure may be included in your application by using the following compiler directive:

```
Option Include WifiScan_t
```

The *auth* member of the `WifiScan_t` structure contains an indicator of the authorization mode of the access point as described in the table below.

Access Point Authorization Mode Values

Value	Meaning
0	No encryption, i.e. an open access point.
1	WEP encryption.
2	WPA/PSK encryption.
3	WPA2/PSK encryption.
4	WPA/WPA2/PSK encryption.

The *rssi* member of the `WifiScan_t` structure gives the relative signal string in units of decibels. The values are generally negative and the less negative values represent stronger signals.

The *mac* member of the `WifiScan_t` structure is an array of bytes giving the MAC address of the access point.

N.B.: This routine must not be called from the `UserInit()` routine.

Wifi.SetConfig

Type Subroutine

Invocation
```
Wifi.SetConfig(ssid)
Wifi.SetConfig(ssid, passwd)
Wifi.SetConfig(ssid, passwd, macAddr)
```

Parameter	Method	Type	Description
ssid	ByVal	String	The access point SSID (30 characters maximum).
passwd	ByVal	String	The access point password (62 characters maximum).
macAddr	ByRef	Byte	The access point MAC address.

Discussion

This subroutine sets the parameters of the access point to which to connect. If a WiFi connection has already been established, it is automatically disconnected.

The first form shown above is for connecting to an open access point, i.e. one that has no password. The third form is used when multiple access points are available having the same SSID and it is desired to connect to a specific one, as distinguished by the MAC address.

Wifi.SetHostname

Type Subroutine

Invocation Wifi.SetHostname(hostname)

Parameter	Method	Type	Description
hostname	ByVal	String	The desired host name (32 characters maximum).

Discussion

This subroutine sets the DHCP hostname for the device.

Wifi.SetIP

Type Subroutine

Invocation Wifi.SetIP(ifc, ipAddr, gwAddr, netmask)

Parameter	Method	Type	Description
ifc	ByRef	anyIntegral	The interface selector.
ipAddr	ByVal	UnsignedLong	The host IP address.
gwAddr	ByVal	UnsignedLong	The gateway IP address.
netMask	ByVal	UnsignedLong	The network mask.

Discussion

This subroutine sets the network parameters for either the Station interface (*ifc* = 0) or the AccessPoint interface (*ifc* = 1). If the network parameters are not set explicitly, the DHCP protocol will be used to obtain an IP address and the other parameters from any available DHCP server.

Wifi.SetMode

Type Subroutine

Invocation Wifi.SetMode(mode)

Parameter	Method	Type	Description
mode	ByVal	anyIntegral	The desired operating mode.

Discussion

This subroutine sets the operating mode of the WiFi interface (see the table below). By default, the ESP8266 will be in Station mode.

WiFi Mode Selector Values

Value	Meaning
1	Station mode.
2	Soft AP mode (currently not supported).
3	Station + Soft AP mode (currently not supported).

Wifi.Status

Type Function returning Byte

Invocation Wifi.Status()

Discussion

This function return the current status of the station interface. The return values are as indicated in the table below.

WiFi Station Status Values

Value	Meaning
0	Station is idle.
1	Station is attempting to connect to the access point.
2	The access point is available but the password was incorrect.
3	The specified access point was not found.
4	Connection to the access point failed.
5	Connection to the access point succeeded.

SPI Flash Allocation

An ESP8266 application usually consists of two binary image files. The first image file, stored at address 0x0000 of the Flash chip contains the application boot code (instruction RAM code, and RAM data) and is limited to 64KB in size. The second image file contains the ROM code (instruction ROM code and read-only data). The primary difference between the handling of the content of the two images is that the content of the first image gets loaded into RAM at boot time while the routines in the second image are loaded into a RAM instruction cache as they are needed or (for read-only data) read directly from Flash when needed.

The ZBasic compiler produces a combined image file containing both of the images described in the previous paragraph. The name of the combined image file is based on the project name but it has the extension `.esp`. The ZBasic compiler structures the application to be compatible with the Flash memory map below.

ESP8266 Memory Map for 512K Flash Chip

Address	Size	Description
0x0000000	0x010000 (64K)	Application boot image
0x0010000	0x001000 (4K)	ZBasic Persistent memory
0x0011000	0x06b000 (428K)	Application ROM code/data
0x007c000	0x004000 (16K)	ESP8266 system parameters

Although most ESP8266 devices come equipped with a 512KB Flash chip, some are available with larger capacity chips, e.g. 1MB, 2MB, 4MB or 8MB. The Flash map for the larger Flash chips is similar to that depicted above. The first three entries have the same start address but the start address for the fourth (the sytem parameter area) is always 16KB from the end of the Flash chip.

The area between the ZBasic Persistent memory and the ESP8266 system parameters is available for use by the application. Any space beyond the end of the application code/data can be used for any purpose. One example of how this "empty" space might be used is a Flash-based file system like SPIFFS (SPI Flash File System). More discussion about SPIFFS is found elsewhere in this document.

Using an SPI Flash File System (SPIFFS)

The ZBasic System Library incorporates a version of Peter Andersson's open source SPIFFS code (available at github.com), slightly modifed and targeted to the ESP8266. The SPIFFS code allows an application to use part of the Flash memory as a single-level (i.e. non-hierarchical) file system. The ZBasic System Library for the ESP8266 contains new routines (e.g. File.Open) to allow interaction with the file system.

In order to use SPIFFS you must designate the starting address and size of the area of Flash that you want to use as the file system. In general, it is advisable to choose the starting address so that the last byte of Flash used in file system is just before the ESP8266 system parameter area which is always located 16KB from the declared end of Flash. So if you wanted a file system 32KB in size the best place to put it is 48KB from the end of Flash. For a 512KB flash that would be at hexadecimal address 0x80000 - 0x4000 - 0x8000 = 0x7c000. Due to the design of the Flash chips used on ESP8266 devices, the file system must begin on a 4KB boundary and be an integral multiple of 4KB in size.

It is important that the Flash memory to be used for SPIFFS be properly initialized. An empty file system can be initialized by erasing the Flash memory, doing so sets all bytes to 0xff. Another useful way to initialize the SPIFFS is to load a prepared SPIFFS image. The ZBasic distribution contains a utility named spiffy.exe that can be used to create and populate a SPIFFS image.

The syntax for invoking spiffy is:

`spiffy [<options>]`

where the available options are as follows.

Option	Description
-h	Display information about invocation options and the exit.
-d<*directory*>	Specify the directory containing files to add to the SPIFFS image. The default directory is files.
-f<*filename*>	Specify the name of the SPIFFS image file. The default image filename is spiff_rom.bin.
-s<*size*>	Specify the size of the SPIFFS image. The default image size is 16KB.

Once the image is built and you've chosen an address for it, it can be programmed into the Flash chip using the esp_tool.exe utility provided with the ZBasic distribution. The esp_tool utility has a substantial feature set including the ability to generate and download application images. The full functionality and invocation options are described in a separate document.