

MRF24WG Universal Driver

User’s Guide

Version 0.5

Internal Use Only

Information contained in this document is confidential and proprietary material of Microchip Technology, Inc. Release, transfer, or sharing of this information outside of Microchip Technology, Inc. without a non-disclosure agreement and permission is strictly prohibited.

[This page left blank]**Table of Contents**

[1 Introduction 8](#_Toc359928984)

[2 Universal Driver Software Overview 9](#_Toc359928985)

[3 Source Code 10](#_Toc359928986)

[3.1 Code Organization 10](#_Toc359928987)

[3.2 Data Types 10](#_Toc359928988)

[4 Stub Functions 11](#_Toc359928989)

[4.1 wf\_gpio\_stub.c 11](#_Toc359928990)

[4.1.1 WF\_GpioInit 11](#_Toc359928991)

[4.1.2 WF\_GpioSetReset 11](#_Toc359928992)

[4.1.3 WF\_GpioSetHibernate 11](#_Toc359928993)

[4.2 wf\_timer\_stub.c 12](#_Toc359928994)

[4.2.1 WF\_TimerInit 12](#_Toc359928995)

[4.2.2 WF\_TimerRead 12](#_Toc359928996)

[4.3 wf\_eint\_stub.c 13](#_Toc359928997)

[4.3.1 WF\_EintInit 13](#_Toc359928998)

[4.3.2 WF\_EintEnable 13](#_Toc359928999)

[4.3.3 WF\_EintDisable 14](#_Toc359929000)

[4.3.4 WF\_isEintDisabled 14](#_Toc359929001)

[4.3.5 External Interrupt Handler 14](#_Toc359929002)

[4.4 wf\_spi\_stub.c 15](#_Toc359929003)

[4.4.1 WF\_SpiInit 15](#_Toc359929004)

[4.4.2 WF\_SpiEnableChipSelect 15](#_Toc359929005)

[4.4.3 WF\_SpiDisableChipSelect 16](#_Toc359929006)

[4.4.4 WF\_SpiTxRx 16](#_Toc359929007)

[4.5 wf\_event\_stub.c 17](#_Toc359929008)

[4.5.1 WF\_ProcessEvent 17](#_Toc359929009)

[4.5.2 WF\_ProcessRxPacket 18](#_Toc359929010)

[5 Driver Customization 19](#_Toc359929011)

[5.1 Error Checking 19](#_Toc359929012)

[5.2 Endianess 19](#_Toc359929013)

[5.3 Inline Code 19](#_Toc359929014)

[5.4 Calculation of WPA Binary Key 19](#_Toc359929015)

[5.5 WPS Security 20](#_Toc359929016)

[5.6 Multicast Filters 21](#_Toc359929017)

[6 Universal Driver API 22](#_Toc359929018)

[6.1 Initialization 22](#_Toc359929019)

[6.1.1 WF\_Init 22](#_Toc359929020)

[6.1.2 WF\_Task 22](#_Toc359929021)

[6.2 Core WiFi Configuration Functions 23](#_Toc359929022)

[6.2.1 WF\_RegionalDomainSet 23](#_Toc359929023)

[6.2.2 WF\_SsidSet 23](#_Toc359929024)

[6.2.3 WF\_NetworkTypeSet 23](#_Toc359929025)

[6.2.4 WF\_ChannelListSet 23](#_Toc359929026)

[6.2.5 WF\_ReconnectModeSet 24](#_Toc359929027)

[6.3 WiFi Security Functions 26](#_Toc359929028)

[6.3.1 WF\_SecurityOpenSet 26](#_Toc359929029)

[6.3.2 WF\_SecurityWepSet 26](#_Toc359929030)

[6.3.2.1 t\_wepContext 26](#_Toc359929031)

[6.3.3 WF\_SecurityWpaSet 27](#_Toc359929032)

[6.3.3.1 t\_wpaContext 27](#_Toc359929033)

[6.3.4 WF\_SecurityWpsSet 28](#_Toc359929034)

[6.3.4.1 t\_wpsContext 28](#_Toc359929035)

[6.3.5 WF\_WpaConvPassphraseToKey 29](#_Toc359929036)

[6.3.6 WF\_WpsKeyGenerate 29](#_Toc359929037)

[6.3.7 WF\_WpsCredentialsGet 30](#_Toc359929038)

[6.3.7.1 t\_wpsCredentials 30](#_Toc359929039)

[6.4 WiFi Connection Functions 31](#_Toc359929040)

[6.4.1 WF\_Connect 31](#_Toc359929041)

[6.4.2 WF\_Disconnect 31](#_Toc359929042)

[6.4.3 WF\_ConnectionStateGet 31](#_Toc359929043)

[6.5 WiFi Scanning Functions 32](#_Toc359929044)

[6.5.1 WF\_Scan 32](#_Toc359929045)

[6.6 WF\_ScanResultGet 32](#_Toc359929046)

[6.6.1.1 t\_ScanResult 32](#_Toc359929047)

[6.7 WiFi Power Save Functions 34](#_Toc359929048)

[6.7.1 WF\_PsPollEnable 34](#_Toc359929049)

[6.7.1.1 t\_psPollContext 34](#_Toc359929050)

[6.7.2 WF\_PsPollDisable 35](#_Toc359929051)

[6.7.3 WF\_Hibernate 35](#_Toc359929052)

[6.7.4 WF\_PowerStateGet 35](#_Toc359929053)

[6.8 WiFi Maximum Power Functions 36](#_Toc359929054)

[6.8.1 WF\_TxPowerFactoryMaxGet 36](#_Toc359929055)

[6.8.2 WF\_TxPowerMaxSet 36](#_Toc359929056)

[6.9 WiFi Status Functions 36](#_Toc359929057)

[6.9.1 WF\_DeviceInfoGet 36](#_Toc359929058)

[6.9.1.1 t\_deviceInfo 36](#_Toc359929059)

[6.9.2 WF\_MacStatsGet 37](#_Toc359929060)

[6.9.2.1 t\_macStats 37](#_Toc359929061)

[6.10 WiFi Multicast Filter Functions 39](#_Toc359929062)

[6.10.1 WF\_HwMulticastFilterSet 39](#_Toc359929063)

[6.10.2 WF\_SwMulticastFilterSet 39](#_Toc359929064)

[6.10.2.1 t\_swMulticastConfig 39](#_Toc359929065)

[6.11 WiFi Packet Tx Functions 41](#_Toc359929066)

[6.11.1 WF\_TxPacketAllocate 41](#_Toc359929067)

[6.11.2 WF\_TxPacketCopy 41](#_Toc359929068)

[6.11.3 WF\_TxPacketTransmit 41](#_Toc359929069)

[6.12 WiFi Packet Rx Functions 42](#_Toc359929070)

[6.12.1 WF\_RxPacketLengthGet 42](#_Toc359929071)

[6.12.2 WF\_RxPacketCopy 42](#_Toc359929072)

[6.12.3 WF\_RxPacketDeallocate 42](#_Toc359929073)

[6.13 WiFi Miscellaneous Functions 43](#_Toc359929074)

[6.13.1 WF\_MacAddressGet 43](#_Toc359929075)

[6.13.2 WF\_MacAddressSet 43](#_Toc359929076)

[6.14 WiFi Advanced Functions 44](#_Toc359929077)

[6.14.1 WF\_ScanContextSet 44](#_Toc359929078)

[6.14.1.1 t\_scanContext 44](#_Toc359929079)

[6.14.2 WF\_AdhocContextSet 45](#_Toc359929080)

[6.14.2.1 t\_adHocNetworkContext 45](#_Toc359929081)

[6.14.3 WF\_TxModeSet 45](#_Toc359929082)

[6.14.4 WF\_BssidSet 46](#_Toc359929083)

[6.14.5 WF\_RssiSet 46](#_Toc359929084)

[6.14.6 WF\_RtsThresholdSet 46](#_Toc359929085)

[6.14.7 WF\_LinkDownThresholdSet 47](#_Toc359929086)

[7 Event Data 48](#_Toc359929087)

[7.1 WF\_EVENT\_INITIALIZATION 48](#_Toc359929088)

[7.2 WF\_EVENT\_CONNECTION\_TEMPORARILY\_LOST 48](#_Toc359929089)

[7.3 WF\_EVENT\_CONNECTION\_FAILED 49](#_Toc359929090)

[7.4 WF\_EVENT\_CONNECTION\_PERMANENTLY\_LOST 51](#_Toc359929091)

[8 Connection Examples 52](#_Toc359929092)

[8.1 Infrastructure Connection, No Security 52](#_Toc359929093)

[8.2 AdHoc Connection, No Security 52](#_Toc359929094)

[8.3 P2P (WiFi Direct) Connection, Push Button 53](#_Toc359929095)

[8.4 P2P (WiFi Direct) Connection, PIN 53](#_Toc359929096)

[9 Appendix 53](#_Toc359929097)

[9.1 Memory Requirements 54](#_Toc359929098)

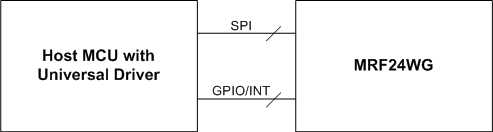
**Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Changes** | **Release Date** | **Author** |
| 0.5 | * Minor edits, corrections * Fixed description of multicast filters | 25 JUNE 2013 | KH |
| 0.4 | * Updated description of *retryCount* parameter in WF\_ReconnectModeSet. * Added description about error checking (Section 5.1) * Added connection examples | 23 JAN 2013 | KH |
| 0.3 | * Added detail about event data * Modified description for wf\_eint\_stub.c * Corrected some cross references * Added event data codes * Some description changes | 29 NOV 2012 | KH |
| 0.2 | * First draft complete | 22 OCT 2012 | KH |
| 0.1 | * Created document | 11 OCT 2012 | KH |

# Introduction

The Universal Driver supports the Microchip MRF24WG WiFi Module and allows one to easily create 802.11 wireless applications. The Universal Driver runs on a Host MCU that connects to the MRF24WG, as shown below.

Figure : Host MCU Connected to MRF24WG



**Features**

* MCU-agnostic; requires only an SPI interface, a timer, 2 GPIO’s, and an interrupt line
* Works with or without a TCP/IP stack
* Works in a ‘round-robin’ or tasking environment
* Customizable via compiler switches to save memory
* Written in portable ‘C’ with all source code provided

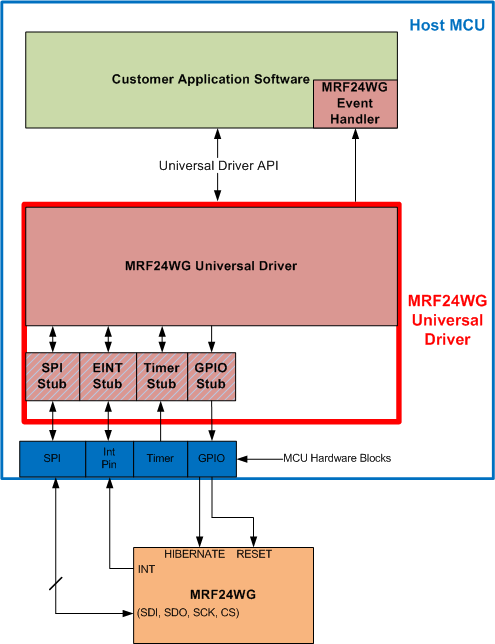
# Universal Driver Software Overview

The Universal Driver (UD) is a collection of ‘C’ modules that fall into one of two categories:

* Modules that do not need to be modified (the ‘core’ driver)
* Modules that contain stub functions specific to MCU hardware and requiring custom code

Below is a diagram showing the partitioning of the Universal Driver software on a MCU. Discussions that follow will reference this diagram.

Figure : Universal Driver on MCU



shows the three major blocks of software that comprise an MRF24WG application. These are:

|  |  |
| --- | --- |
| **Customer Application Software** | This is the application code, and may include a TCP/IP stack. Note that an MRF24WG Event Handler is part of this block. The event handler module takes the form of two callback function:  () – notifies application of WiFi-related events  () – notifies application that an Rx packet was received |
| **MRF24WG Universal Driver Software** | This is the core driver code and is supplied by Microchip. It should not need to be modified. |
| **Stub Software** | The function prototypes are provided by Microchip. The Universal Driver will call these functions. The stub functions control MUC-specific hardware:   * SPI Interface * GPIO control * 1ms Timer * Interrupt (from MRF24WG) |

# Source Code

## Code Organization

The directory structure for the Universal Driver is:

mrf24wg\_universal\_driver - top level directory

ud\_inc - contains include subdirectories

internal - includes only needed by core driver

shared - includes needed by application and core driver

ud\_src - driver source code

stub\_templates - modules containing empty stub functions

## Data Types

The Universal Driver source code uses data types as defined in stdint.h and stdbool.h (e.g. uint8\_t, uint16\_t, uint32\_t, bool). In fact, the header file wf\_universal\_driver.h includes these standard header files. If your toolchain does not support stdint.h and stdbool.h then remove these includes and create typedef’s for the data types. For example:

typedef unsigned char uint8\_t

typedef unsigned short uint16\_t

typedef unsigned long uint32\_t

typedef unsigned char bool

#define false 0

#define true 1

# Stub Functions

Before one can use the Universal Driver API there are some MCU-specific stub functions that must be coded. The Universal Driver will call these functions during runtime.

This section covers in detail the stub functions that are required by the Universal Driver, and must be customized for the Host MCU. There are five stub modules, each of which are discussed in this section. Also, there is an example demo provided in the release of all the stub functions coded for a PIC32MX795F512L MCU.

All define constants used in this section are in the wf\_universal\_driver.h file.

## wf\_gpio\_stub.c

This module contains stub functions to control the HIBERNATE and RESET pins on the MRF24WG.

### WF\_GpioInit

|  |  |
| --- | --- |
| Prototype | void **WF\_Gpio\_Init**(void); |
| Description | Configures the GPIO pins that will be used to control the MRF24WG RESET and HIBERNATE pins. The function should:  Set the HIBERNATE pin level high and configure it as an output  Set the RESET pin low and configure it as an output |
| Inputs | None |
| Returns | None |

### WF\_GpioSetReset

|  |  |
| --- | --- |
| Prototype | void **WF\_Gpio\_SetReset**(uint8\_t level); |
| Description | Sets the MRF24WG RESET pin high or low |
| Inputs | level -- WF\_HIGH or WF\_LOW |
| Returns | None |

### WF\_GpioSetHibernate

|  |  |
| --- | --- |
| Prototype | void **WF\_Gpio\_SetHibernate**(uint8\_t level); |
| Description | Sets the MRF24WG HIBERNATE pin high or low |
| Inputs | level -- WF\_HIGH or WF\_LOW |
| Returns | None |

## wf\_timer\_stub.c

This module contains stub functions that control a 1ms timer needed by the Universal Driver.

### WF\_TimerInit

|  |  |
| --- | --- |
| Prototype | void **WF\_TimerInit**(void); |
| Description | Configures and starts timer that increments a 32-bit counter every 1ms. If this requires the creation of a timer interrupt it should perform the following actions:   1. increment 1ms counter 2. clear the interrupt |
| Inputs | None |
| Returns | None |

### WF\_TimerRead

|  |  |
| --- | --- |
| Prototype | uint32\_t **WF\_TimerRead**(void); |
| Description | Returns the current value of the 32-bit 1ms counter. The following pseudo-code is suggested to guarantee the counter value is not corrupted during the read by a timer interrupt occurring while this function is executing. For example, a PIC32 could implement this function as shown below:  uint32\_t WF\_TimerRead(void)  {  uint32\_t retValue;    IEC0CLR = \_IEC0\_T1IE\_MASK; // Disable Timer1 interrupt  retValue = g\_msCounter; // safely read the global counter value  IEC0SET = \_IEC0\_T1IE\_MASK; // Enable Timer1 interrupt  return retValue;  } |
| Inputs | None |
| Returns | 1ms counter value |

## wf\_eint\_stub.c

This module contains stub functions that deal with the MRF24WG external interrupt. These functions should not be called from the application code.

### WF\_EintInit

|  |  |
| --- | --- |
| Prototype | void **WF\_EintInit**(void); |
| Description | Configures the MRF24WG external interrupt and enables it. Typically this involves configuring the interrupt pin on the MCU as either a falling edge triggered interrupt. |
| Inputs | None |
| Returns | None |

### WF\_EintEnable

|  |  |
| --- | --- |
| Prototype | void **WF\_EintEnable**(void); |
| Description | Enables the MRF24WG external interrupt. During operations, the Universal Driver will, for very short periods, disable the external interrupt. A falling edge may occur while the interrupt is disabled. If that happens, the system will never get another interrupt after the interrupt is reenabled. Because of this, the pseudo-code for this function is:  // if interrupt line is low (may have missed the falling edge)  IF is low  Force (trigger) the Eint interrupt  ENDIF  Enable Eint  Example code for a PIC32 is:  void **WF\_EintEnable**(void)  {  // if interrupt line is low  if ( INT1\_IO == 0 )  {  // force the interrupt as soon as it is enabled below  IFS0SET = INT1\_MASK;  }  // enable the external interrupt  IEC0SET = INT1\_MASK;  } |
| Inputs | None |
| Returns | None |

### WF\_EintDisable

|  |  |
| --- | --- |
| Prototype | void **WF\_EintDisable**(void); |
| Description | Disables the MRF24WG external interrupt. |
| Inputs | None |
| Returns | None |

### WF\_isEintDisabled

|  |  |
| --- | --- |
| Prototype | bool **WF\_isEintDisabled**(void); |
| Description | Returns true if the interrupt is currently disabled, else returns false |
| Inputs | None |
| Returns | True or false |

### External Interrupt Handler

The actual interrupt handler can have any name as the Universal Driver does not directly call it. The interrupt priority should be relatively high. The interrupt handler must do the following:

1. Clear the interrupt
2. Disable the interrupt
3. Call WF\_EintHandler()

When the Eint interrupt exits, it must be disabled. The Universal Driver will reenable it. An example PIC32 interrupt handler is:

void \_\_attribute((interrupt(ipl3), vector(\_EXTERNAL\_1\_VECTOR), nomips16)) \_WFInterrupt(void)

{

IFS0CLR = INT1\_MASK; // clear the external interrupt

IEC0CLR = INT1\_MASK; // disable external interrupt

WF\_EintHandler(); // call Universal Driver handler function

}

## wf\_spi\_stub.c

The Universal Driver uses an SPI interface to communicate with the MRF24WG. SPI communications have four possible modes, and the MRF24WG supports two of them, Modes 0 and 3, as shown in Table 1below.

Table : Supported SPI Modes

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SPI Mode** | **CPOL** | **CPHA** | **Description** | **Supported by MRF24WG** |
| 0 | 0 | 0 | Idle state for clock is low, active state is high  Read data captured on clocks rising edge  Write data propagated on clocks falling edge | Yes |
| 1 | 0 | 1 | Idle state for clock is low, active state is high  Read data captured on clocks falling edge  Write data propagated on clocks rising edge | No |
| 2 | 1 | 0 | Idle state for clock is high, active state is low  Read data captured on clocks falling edge  Write data propagated on clocks rising edge | No |
| 3 | 1 | 1 | Idle state for clock is high, active state is low  Read data captured on clocks rising edge  Write data propagated on clocks falling edge | Yes |

### WF\_SpiInit

|  |  |
| --- | --- |
| Prototype | void **WF\_SpiInit**(void); |
| Description | Configures the SPI interface with the following:   * SPI master * Maximum SPI clock rate is 25MHz * SPI interrupts should be disabled * Mode 0 or 3 (see ) * MRF24WG chip select line high (disabled) * SPI controller enabled |
| Inputs | None |
| Returns | None |

### WF\_SpiEnableChipSelect

|  |  |
| --- | --- |
| Prototype | void **WF\_SpiEnableChipSelect**(void); |
| Description | Sets the MRF24WG SPI chip select line low.  If the SPI controller is being shared with another device that uses a different SPI mode then save the default SPI context in this function and configure the MRF24WG SPI mode. |
| Inputs | None |
| Returns | None |

### WF\_SpiDisableChipSelect

|  |  |
| --- | --- |
| Prototype | void **WF\_SpiDisableChipSelect**(void); |
| Description | Sets the MRF24WG SPI chip select line high.  If the SPI controller is being shared with another device that uses a different SPI mode then restore the default SPI context that was saved in WF\_SpiEnableChipSelect. |
| Inputs | None |
| Returns | None |

### WF\_SpiTxRx

|  |  |
| --- | --- |
| Prototype | void **WF\_SpiTxRx**(const uint8\_t \*p\_txBuf,  uin16\_t txLength,  uint8\_t \*p\_rxBuf,  uin16\_t rxLength); |
| Description | This function transmits and receives data from the MRF24WG. The Universal Driver calls prior to calling ; and may be called multiple times before is called.  If rxLength is > txLength then write out filler bytes of 0x00 in order to get all the read bytes.  If txLength > rxLength then throw away the extra read bytes. Do not write the extra read bytes to p\_rxBuf. |
| Inputs | p\_txBuf – pointer to the transmit buffer  txLength – number of bytes to be transmitted  p\_rxBuf – pointer to receive buffer  rxLength – number of bytes to be received |
| Returns | None |

## wf\_event\_stub.c

This stub module is not MCU-specific, but it does contain two stub functions that must be modified for the application.

### WF\_ProcessEvent

|  |  |
| --- | --- |
| Prototype | void **WF\_ProcessEvent**(uin8\_t eventType, uint32\_t eventData); |
| Description | This function is called by the Universal Driver when a WiFi event has occurred. The application code should set up a switch statement and process events as needed. |
| Inputs | eventType – The type of event (see )  eventData – Additional data about the event (not used for all events) |
| Returns | None |

Table : Universal Driver Event Types

| **eventType** | **Description** |
| --- | --- |
| WF\_EVENT\_INITIALIZATION | This event signifies the success or failure of the MRF24WG initialization process. See Section for *eventData* values. |
| WF\_EVENT\_CONNECTION\_SUCCESSFUL | This event signifies the successful connection to an Access Point. *eventData* is not used. |
| WF\_EVENT\_CONNECTION\_TEMPORARILY\_LOST | This event signifies that the current WiFi connection has been temporarily lost (typically due to a beacon timeout) and the MRF24WG is trying to reestablish the connection. See Section for *eventData* values. |
| WF\_EVENT\_CONNECTION\_REESTABLISHED | This event signifies that the WiFi connection that was temporarily lost has been reestablished. *eventData* is not used. |
| WF\_EVENT\_CONNECTION\_PERMANENTLY\_LOST | This event signifies that the WiFi connection has been permanently lost and the MRF24WG is not trying to reestablish the connection. See Section for *eventData* values. See Section for *eventData* values. |
| WF\_EVENT\_CONNECTION\_FAILED | This event signifies that the MRF24WG was unable to establish a connection. See Section for *eventData* values. |
| WF\_EVENT\_SCAN\_RESULTS\_READY | This event signifies that WiFi scan results are ready to read from the MRF24WG. *eventData* contains the number of scan results. |
| WF\_WPS\_EVENT\_KEY\_CALCULATION\_REQUEST | This event signifies that that, during the WPS connection process, the MRF24WG has sent the WPA/WPA2 ASCII passphrase to the Host MCU. The host should calculate the binary key and send it back to the MRF24WG. *eventData* is not used. |
| WF\_EVENT\_MRF24WG\_MODULE\_ASSERT | This event signifies the MRF24WG has hit an assert condition.  *eventData*[23:16] – Module Number  *eventData*[7:0] – Assert code |
| WF\_EVENT\_ERROR | This event signifies the Universal Driver has detected an error condition. See *t\_udEventErrors* in wf\_events.h. |

### WF\_ProcessRxPacket

|  |  |
| --- | --- |
| Prototype | void **WF\_ProcessRxPacket**(void); |
| Description | This function is called by the Universal Driver when an Rx Ethernet packet has been received. See Section 6.12 for how to process incoming packets. |
| Inputs | None |
| Returns | None |

# Driver Customization

The Universal Driver has several parameters that can be customized, and are described in this section. All of the #defines (except one) discussed in this section can be found in wf\_customize.h.

## Error Checking

The define WF\_ERROR\_CHECKING, which is not in wf\_customize.h, is used to enable or disable Universal Driver error checking. It is recommended that WF\_ERROR\_CHECKING be defined at the project level when bringing up the driver. See  *t\_udEventErrors* in wf\_events.h for a description of the possible error codes.

## Endianess

One of two defines must be declared to define the Host MCU as LITTLE\_ENDIAN or BIG\_ENDIAN. The demo code provided has this declaration:

#define LITTLE\_ENDIAN 1234

Normally, it is only necessary to declare:

#define LITTLE\_ENDIAN

The addition of the1234 for the demo is to match the lwIP TCP/IP stack header file to eliminate compile warnings. Note that the MRF24WG is big-endian.

## Inline Code

The Universal Driver declares several functions as INLINE. Define INLINE as appropriate for the toolchain being used. For example, the PIC XC32 compiler has the ‘inline’ keyword, so this declaration would be used:

#define INLINE inline

If, for whatever reason, you wish to disable the driver from using inline functions simply declare:

#define INLINE

## Calculation of WPA Binary Key

When using WPA/WPA2 security, and one needs to convert an ASCII passphrase/SSID to a binary key there are two choices:

1. The Host MCU can calculate the binary key (4 seconds on PIC32, but takes approx. 18Kbytes of code)
2. The MRF24WG can calculate the binary key (30 seconds)

The define to comment in or out is: WF\_USE\_HOST\_WPA\_KEY\_CALCULATION

To summarize:

* Comment out this define if the application does not need to convert a WPA ASCII key and SSID to a binary key
* Comment out this define if it is desired to have the MRF24WG calculate the key
* Add this define if the Host MCU should calculate WPA binary keys

## WPS Security

If the application will not be using WPS security, then comment out WF\_USE\_WPS\_SECURITY to save code and data space.

## Multicast Filters

The MRF24WG supports two types of multicast filtering – hardware and software. If doing multicast filtering, only one of these can be selected.

There are two defines that control which of multicast mode is selected:

#define WF\_USE\_HARDWARE\_MULTICAST\_FILTER

#define WF\_USE\_SOFTWARE\_MULTICAST\_FILTER

If multicast filtering is required, then select one of the above defines and comment the other one out. If multicast filtering is not required by the application than comment out both defines to save code space.

More information about multicast filters can be found in (section 6.10).

# Universal Driver API

This section covers all Universal Driver functions that the application can call. The general calling sequence for most applications will be:

## Initialization

The functions in this section are used to initialize the Universal Driver, reset the MRF24WG, and prepare for connectivity.

### WF\_Init

|  |  |
| --- | --- |
| Prototype | void **WF\_Init**(void); |
| Description | This function must be called before any other functions in the API.  It starts the process of resetting the MRF24WG and initializing the Universal Driver. After calling this function, must be called periodically for the initialization to complete. The event will occur upon completion of the WiFi initialization; no other Universal Driver functions should be called until after this event. |
| Inputs | None |
| Returns | None |

### WF\_Task

|  |  |
| --- | --- |
| Prototype | void **WF\_Task**(void); |
| Description | After calling , this function must be called periodically for the Universal Driver to function. |
| Inputs | None |
| Returns | None |

## Core WiFi Configuration Functions

The functions in this section perform the basic WiFi configuration, and must always be called before connecting.

### WF\_RegionalDomainSet

|  |  |
| --- | --- |
| Prototype | void **WF\_RegionalDomainSet**(uint8\_t regionalDomain); |
| Description | Configures the MRF24WG to a specific regional domain. |
| Inputs | regionalDomain – one of the following:  WF\_DOMAIN\_FCC (channels 1-11)  WF\_DOMAIN\_ETSI (channels 1-13)  WF\_DOMAIN (channels 1-14)  WF\_DOMAIN\_OTHER (channels 1-14) |
| Returns | None |

### WF\_SsidSet

|  |  |
| --- | --- |
| Prototype | void **WF\_SsidSet**(uint8\_t \*p\_ssid, uint8\_t ssidLength); |
| Description | Sets the network name of the network being joined. The SSID must be 32 characters or less. If using a string, the NULL character is not counted. |
| Inputs | p\_ssid – pointer to the SSID  ssidLength – number of bytes in the SSID |
| Returns | None |

### WF\_NetworkTypeSet

|  |  |
| --- | --- |
| Prototype | void **WF\_NetworkTypeSet**(uint8\_t networkType); |
| Description | Selects the type of WiFi network. |
| Inputs | networkType – one of the following:  WF\_NETWORK\_TYPE\_INFRASTRUCTURE  WF\_NETWORK\_TYPE\_ADHOC  WF\_NETWORK\_TYPE\_P2P  WF\_NETWORK\_TYPE\_SOFT\_AP |
| Returns | None |

### WF\_ChannelListSet

|  |  |
| --- | --- |
| Prototype | void **WF\_ChannelListSet**(uint8\_t \*p\_channelList, uint8\_t numChannels); |
| Description | Sets the channel list that MRF24WG should use when scanning or connecting. |
| Inputs | p\_channelList – pointer to list of channels  numChannels – number of channels in the list |
| Returns | None |

### WF\_ReconnectModeSet

|  |  |
| --- | --- |
| Prototype | void **WF\_ReconnectModeSet**(uint8\_t retryCount,  uint8\_t deauthAction,  uint8\_t beaconTimeout,  uint8\_t beaconTimoutAction); |
| Description | This function controls how the MRF24WG behaves when an existing WiFi connection is lost. The MRF24WG can lose an existing connection in one of two ways:   * Beacon timeout * Deauth received from AP   There are two options with respect to regaining a lost WiFi connection:   1. MRF24WG informs the host that the connection was temporarily lost and then the MRF24WG retries N times (or forever) to regain the connection. 2. MRF24WG simply informs the host application that the connection is lost, and it is up to the host to regain the connection via the API.   Note: the *retryCount* parameter also applies when initially connecting. That is, the *retryCount* tells the MRF24WG how many time to try to connect to a WiFi network before giving up and generating the event. |
| Inputs | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | retryCount – | Number of times the MRF24WG should try to regain the connection. Also, see note in description.   |  |  | | --- | --- | | 0 | Do not try to regain a connection (simply report event to host) | | 1:254 | Number of times to try to regain the connection | | 255 | MRF24WG will retry forever (do not use for AdHoc connections) | | | deauthAction – | In the event of a deauth from the AP, the action the MRF24WG should take:  WF\_ATTEMPT\_TO\_RECONNECT or  WF\_DO\_NOT\_ATTEMPT\_TO\_RECONNECT | | beaconTimeout – | Number of missed beacons before the MRF24WG designates the connection as lost.   |  |  | | --- | --- | | 0 | MRF24WG will **not** monitor the beacon timeout condition | | 1:255 | Number of missed beacons before designating the connection as lost. | | | beaconTimeoutAction – | In the event of a beacon timeout, the action the MRF24WG should take:  WF\_ATTEMPT\_TO\_RECONNECT or  WF\_DO\_NOT\_ATTEMPT\_TO\_RECONNECT | |
| Examples | Below are several examples of how this function could be used. Example 1 is probably the most common usage.  **Example 1**: MRF24WG should retry forever if either a deauth or beacon timeout occurs (beacon timeout is 3 beacon periods):  WF\_SetReconnectMode(WF\_RETRY\_FOREVER,  WF\_ATTEMPT\_TO\_RECONNECT,  3,  WF\_ATTEMPT\_TO\_RECONNECT);  **Example 2**: MRF24WG should not do any connection retries and only report deauth events to the host.  WF\_SetReconnectMode(0,  WF\_DO\_NOT\_ATTEMPT\_TO\_RECONNECT,  0,  WF\_DO\_NOT\_ATTEMPT\_TO\_RECONNECT);  **Example 3**: MRF24WG should not do any connection retries, but report deauth and beacon timeout events to host. Beacon timeout should be 5 beacon periods.  WF\_SetReconnectMode(0,  WF\_DO\_NOT\_ATTEMPT\_TO\_RECONNECT,  5,  WF\_DO\_NOT\_ATTEMPT\_TO\_RECONNECT);  **Example 4**: MRF24WG should ignore beacon timeouts, but attempt to reconnect 3 times if a deauth occurs.  WF\_SetReconnectMode(3,  WF\_ATTEMPT\_TO\_RECONNECT,  0,  WF\_DO\_NOT\_ATTEMPT\_TO\_RECONNECT); |
| Returns | None |

## WiFi Security Functions

There are four security modes for a WiFi connection – open (no security), WEP, WPA, or WPS. One of security types must be selected before connecting.

### WF\_SecurityOpenSet

|  |  |
| --- | --- |
| Prototype | void **WFSecurityOpenSet**(void); |
| Description | Sets the WiFi security to none. |
| Inputs | None |
| Returns | None |

### WF\_SecurityWepSet

|  |  |
| --- | --- |
| Prototype | void **WF\_SecurityWepSet**(t\_wepContext \*p\_context); |
| Description | Sets the connection security to WEP |
| Inputs | p\_context – pointer to WEP context structure (see Section 6.3.2.1) |
| Returns | None |

#### t\_wepContext

typedef struct wepContext

{

uint8\_t wepSecurityType;

uint8\_t wepKeyIndex;

uint8\_t wepKey[WF\_MAX\_WEP\_KEY\_LENGTH];

uint8\_t wepKeyLength;

uint8\_t wepKeyType;

} **t\_wepContext**;

|  |  |
| --- | --- |
| **Field** | **Description** |
| wepSecurityType | WF\_SECURITY\_WEP\_40 or WF\_SECURITY\_WEP\_104 |
| wepKeyIndex | The default key to use 0 (typical) thru 3 |
| wepKey | Array containing four WEP binary keys. This will be four, 5-byte keys for WEP-40 or four, thirteen-byte keys for WEP-104. |
| wepKeyLength | Number of bytes in wepKey |
| wepKeyType | WF\_SECURITY\_WEP\_OPENKEY(typical) or WF\_SECURITY\_WEP\_SHAREDKEY |

### WF\_SecurityWpaSet

|  |  |
| --- | --- |
| Prototype | void **WF\_SecurityWpaSet**(t\_wpaContext \*p\_context); |
| Description | Sets the connection security to WPA |
| Inputs | p\_context – pointer to WPA context structure (see Section 6.3.3.1) |
| Returns | None |

#### t\_wpaContext

The t\_wpaContext structure contains another structure, t\_wpaKeyInfo, also described here.

typedef struct

{

uint8\_t key[WF\_MAX\_PASSPHRASE\_LENGTH];

uint8\_tkeyLength**;**

uint8\_t ssid[WF\_MAX\_SSID\_LENGTH];

uint8\_t ssidLen;

} **t\_wpaKeyInfo**;

typedef struct wpaContext

{

uint8\_t wpaSecurityType;

t\_wpaKeyInfo keyInfo;

} **t\_wpaContext**;

Table : t\_wpaContext Fields

|  |  |
| --- | --- |
| **Fields** | **Description** |
| wpaSecurityType | Selects WPA or WPA2 security. There is also an auto-select feature described below. In addition, the format of the key (ASCII or binary) is indicated. Binary keys are used directly; ASCII keys must be converted to binary keys, either by the host or the MRF24WG.     |  |  | | --- | --- | | **#define Value** | **Description** | | WF\_SECURITY\_WPA\_WITH\_KEY | **Binary** PSK (Pre-shared Key) key will be provided for **WPA** security. | | WF\_SECURITY\_WPA\_WITH\_PASS\_PHRASE | **ASCII** passphrase will be provided for **WPA** security | | WF\_SECURITY\_WPA2\_WITH\_KEY | **Binary** PSK key will be provided for **WPA2** security. | | WF\_SECURITY\_WPA2\_WITH\_PASSPHRASE | **ASCII** passphrase will be provided for **WPA** security | | WF\_SECURITY\_WPA\_AUTO\_WITH\_KEY | Same as *WF\_SECURITY\_WPA\_WITH\_KEY* or *WF\_SECURITY\_WPA2\_WITH\_KEY* except MRF24WG will connect to the AP using highest level security the AP supports (WPA or WPA2). | | WF\_SECURITY\_WPA\_AUTO\_WITH\_PASSPHRASE | Same as *WF\_SECURITY\_WPA\_WITH\_PASS\_PHRASE* or *WF\_SECURITY\_WPA2\_WITH\_PASS\_PHRASE* except MRF24WG will connect to the AP using highest level security the AP supports (WPA or WPA2). | |
| keyInfo | See Table 4 below. |

Table : t\_wpaKeyInfo Fields

|  |  |
| --- | --- |
| **Fields** | **Description** |
| key | This is either the binary key or the ASCII passphrase. Binary keys must be 32 bytes in length. ASCII passphrases must be between 8 and 63 bytes, and do not include a NULL terminator. |
| keyLength | Number of bytes in the *key* array. |
| ssid | The network SSID. This is only needed if *key* is an ASCII passphrase, as a WPA binary key is calculated from both the passphrase and the SSID. If not using set to NULL. |
| ssidLength | The length of *ssid*. Set to 0 if not being used. |

### WF\_SecurityWpsSet

|  |  |
| --- | --- |
| Prototype | void **WF\_SecurityWpsSet**(t\_wpsContext \*p\_context); |
| Description | Sets the connection security to WPS |
| Inputs | p\_context – pointer to WPS context structure (see Section 6.3.4.1) |
| Returns | None |

#### t\_wpsContext

typedef struct wpsContext

{

uint8\_t wpsSecurityType;

uint8\_t wpsPin[WF\_WPS\_PIN\_LENGTH];

uint8\_t wpsPinLength;

#if defined(WF\_USE\_HOST\_WPA\_KEY\_CALCULATION)

bool getPassPhrase;

t\_wpaKeyInfo \*p\_keyInfo;

#endif /\* WF\_USE\_HOST\_WPA\_KEY\_CALCULATION \*/

} **t\_wpsContext**;

Table : t\_wpsContext Fields

|  |  |
| --- | --- |
| **Fields** | **Description** |
| wpsSecurityType | Either WF\_SECURITY\_WPS\_PUSH\_BUTTON or WF\_SECURITY\_WPS\_PIN |
| wpsPin | If *wpsSecurityType* is WF\_SECURITY\_WPS\_PIN then the WPS pin must be defined here. WPS pin is always 8-digits, seven digits for the actual pin, and the last digit is the checksum. |
| wpsPinLength | If *wpsSecurityType* is WF\_SECURITY\_WPS\_PIN then this must set to 8. |
| getPassPhrase | This field is only compiled in when WF\_USE\_HOST\_WPA\_KEY\_CALCULATION is defined.  If True:  The MRF24WG will send the ASCII passphrase back to the host so the host MCU can (more quickly) calculate the binary key. The Host MCU will be notified of this via the WF\_WPS\_EVENT\_KEY\_CALCULATION\_REQUEST event (see Section 4.5.1).  If False:  The MRF24WG will not send the ASCII passphrase back to the host; instead, it (the MRF24WG) will perform the binary key calculation. |
| p\_keyInfo | This field is only compiled in when WF\_USE\_HOST\_WPA\_KEY\_CALCULATION is defined.  This is a pointer to where Universal Driver will store passphrase information it receives from the MRF24WG. This must be in global memory, not on the stack. See Section 4.5.1 for how this structure is used. |

### WF\_WpaConvPassphraseToKey

|  |  |
| --- | --- |
| Prototype | void **WF\_WpaConvPassphraseToKey**(t\_wpaKeyInfo \*p\_keyInfo); |
| Description | Converts an ASCII passphrase and SSID into a WPA binary key. |
| Inputs | p\_keyInfo – See 6.3.3.1.  **Important Note**:  This function will overwrite two fields in the input structure. Specifically:  p\_keyInfo->key will contain the binary key (the ASCII key will be overwritten).  p\_keyInfo->keyLength will set to 32, the length of the binary key |
| Returns | None |

### WF\_WpsKeyGenerate

|  |  |
| --- | --- |
| Prototype | void **WF\_WpsKeyGenerate**(void); |
| Description | This function is used in conjunction with the WF\_WPS\_EVENT\_KEY\_CALCULATION\_REQUEST event. See Section 4.5.1 for more information. |
| Inputs | None |
| Returns | None |

### WF\_WpsCredentialsGet

|  |  |
| --- | --- |
| Prototype | void **WF\_WpsGetCredentials**(t\_wpsCredentials \*p\_cred); |
| Description | Retrieves the WPS credentials from the MRF24WG |
| Inputs | p\_cred |
| Returns | None |

#### t\_wpsCredentials

typedef struct wpsCredentialsStruct

{

uint8\_t ssid[WF\_MAX\_SSID\_LENGTH];

uint8\_t netKey[WF\_MAX\_PASSPHRASE\_LENGTH];

uint16\_t authType;

uint16\_t encType;

uint8\_t netIdx;

uint8\_t ssidLen;

uint8\_t keyIdx;

uint8\_t keyLen;

uint8\_t bssid[WF\_MAC\_ADDRESS\_LENGTH];

} t\_wpsCredentials;

|  |  |
| --- | --- |
| **Fields** | **Description** |
| ssid | Network SSID |
| netKey | Binary security key (if security is not open) |
| authType | One of the following:  WF\_AUTH\_OPEN  WF\_AUTH\_WPA\_PSK  WF\_AUTH\_SHARED  WF\_AUTH\_WPA  WF\_AUTH\_WPA2  WF\_AUTH\_WPA2\_PSK |
| encType | One of the following:  WF\_ENC\_NONE  WF\_ENC\_WEP  WF\_ENC\_TKIP  WF\_ENC\_AES |
| netIdx | Not used |
| ssidLen | Number of bytes in *ssid* |
| keyIdx | Only valid encType = WF\_ENC\_WEP. This is the index of the WEP key being used. |
| keyLen | Number of bytes in *netKey* |
| Bssid | MAC address of AP |

## WiFi Connection Functions

### WF\_Connect

|  |  |
| --- | --- |
| Prototype | void **WF\_Connect**(void); |
| Description | Directs the MRF24WG to connect to a WiFi network. Upon a success connection the WF\_EVENT\_CONNECTION\_SUCCESSFUL event occurs. If the MRF24WG uses up all it retries, or gets rejected by the network WF\_EVENT\_CONNECTION\_FAILED event occurs. |
| Inputs | None |
| Returns | None |

### WF\_Disconnect

|  |  |
| --- | --- |
| Prototype | void **WF\_Disconnect**(void); |
| Description | Forces the MRF24WG to drop the current connection. No event is generated when a connection is terminated via this function call. |
| Inputs | None |
| Returns | None |

### WF\_ConnectionStateGet

|  |  |
| --- | --- |
| Prototype | void **WF\_ConnectionStateGet**(uint8\_t \*p\_state); |
| Description | Retrieves the current state of the connection |
| Inputs | p\_state – pointer where the connection state will be written. State values are:   |  | | --- | | WF\_CSTATE\_NOT\_CONNECTED | | WF\_CSTATE\_CONNECTION\_IN\_PROGRESS | | WF\_CSTATE\_CONNECTED\_INFRASTRUCTURE | | WF\_CSTATE\_CONNECTED\_ADHOC | | WF\_CSTATE\_RECONNECTION\_IN\_PROGRESS | | WF\_CSTATE\_CONNECTION\_PERMANENTLY\_LOST | |
| Returns | None |

## WiFi Scanning Functions

### WF\_Scan

|  |  |
| --- | --- |
| Prototype | void **WF\_Scan**(uint8\_t scanMode); |
| Description | Requests the MRF24WG to perform a scan using the current channel list. When the scan is complete the WF\_EVENT\_SCAN\_RESULTS\_READY event occurs. |
| Inputs | scanMode – WF\_SCAN\_FILTERED or WF\_SCAN\_ALL  WF\_SCAN\_FILTERED:   * If SSID defined only scan results with that SSID are retained * If SSID not defined all scanned SSID’s will be retained * Only scan results from Infrastructure or AdHoc networks are retained, depending on the configured network type * The only channels scanned are those set in WF\_SetChannelList()   WF\_SCAN\_ALL:   * Can be called immediately after WF\_EVENT\_INITIALIZATION event * All scan results are retained (both Infrastructure and Ad Hoc networks) * All channels within the MRF24W’s regional domain will be scanned |
| Returns | None |

## WF\_ScanResultGet

|  |  |
| --- | --- |
| Prototype | void **WF\_ScanResultGe**(uint8\_t listIndex, t\_scanResult \*p\_scanResult); |
| Description | After the WF\_EVENT\_SCAN\_RESULTS\_READ event occurs this function can be called to retrieve the results. |
| Inputs | listIndex – index into the list of scan results  t\_scanResult – pointer where scan result will be written. See . |
| Returns | None |

#### t\_ScanResult

typedef struct

{

uint8\_t bssid[WF\_BSSID\_LENGTH];

uint8\_t ssid[WF\_MAX\_SSID\_LENGTH];

uint8\_t apConfig;

uint8\_t reserved;

uint16\_t beaconPeriod;

uint16\_t atimWindow;

uint8\_t basicRateSet[WF\_MAX\_NUM\_RATES];

uint8\_t rssi;

uint8\_t numRates;

uint8\_t DtimPeriod;

uint8\_t bssType;

uint8\_t channel;

uint8\_t ssidLen;

} t\_scanResult;

|  |  |
| --- | --- |
| **Fields** | **Description** |
| bssid | Network BSSID |
| ssid | Network SSID |
| apConfig | Access Point configuration   |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | WPA2 | WPA | Preamble | Privacy | Reserved | Reserved | Reserved | IE |  |  |  | | --- | --- | | IE | 0: AP is not broadcasting any Information Elements  1: AP is broadcasting 1 or more Information Elements | | Privacy | 0: AP is open (no security)  1: AP is using security. If fields *WPA* and *WPA2* are 0, then WEP security | | Preamble | 0: AP transmitting with short preamble  1: AP transmitting with long preamble | | WPA | Only valid if *privacy* is 1.  0: AP does not support WPA  1: AP does support WPA | | WPA2 | Only valid if *privacy* is 1.  0: AP does not support WPA2  1: AP does support WPA2 | |
| beaconPeriod | Network beacon interval, in ms |
| atimWindow | Only valid if *bssType* is WF\_INFRASTRUCTURE. Announcement Traffic Indication Time in ms. |
| basicRateSet | List of network basic rates. Each byte has the following format:  Bit 7  0: rate is not part of basic rate set  1: rates is part of basic rate set  Bits 6:0  Multiply value by 500kbps to get supported rate. |
| rssi | Signal strength received from frame beacon or probe response |
| numRates | Number of values in *basicRateSet* |
| dtimPeriod | Part of TIM element |
| bssType | WF\_INFRASTRUCTURE or WF\_ADHOC |
| channel | Channel number |
| ssidLen | Number of characters in *ssid* |

## WiFi Power Save Functions

### WF\_PsPollEnable

|  |  |
| --- | --- |
| Prototype | void **WF\_PsPollEnable**(t\_psPollContext \*p\_context); |
| Description | Enables PS-Poll mode. PS-Poll (Power-Save Poll) is a mode allowing for longer battery life. The MRF24WG coordinates with the Access Point to go to sleep and wake up at periodic intervals to check for messages, which the Access Point will buffer. By default, PS-Poll mode is disabled.  When PS Poll is enabled, the WF Host Driver will automatically force the MRF24WG to wake up each time the Host sends Tx data or a control message to the MRF24WG. When the Host message transaction is complete the MRF24WG driver will automatically reactivates PS Poll mode.  **Import Note:**  When the application is likely to experience a high volume of data traffic then PS-Poll mode should be disabled for two reasons:   1. No power savings will be realized in the presence of heavy data traffic. 2. Performance will be impacted adversely as the WiFi Host Driver continually activates and deactivates PS-Poll mode via SPI messages. |
| Inputs | p\_context – see Section 6.7.1.1 |
| Returns | None |

#### t\_psPollContext

typedef struct psPollContext

{

uint16\_t listenInterval;

uint16\_t dtimInterval;

bool useDtim;

} t\_psPollContext;

| **Fields** | **Description** |
| --- | --- |
| listenInterval | Number of 100ms intervals between instances when the MRF24WG wakes up to received buffered messages from the network. Typically the listen interval is 1,  Range:   |  |  | | --- | --- | | 1 | MRF24WG wakes up every 100ms to receive buffered messages | | 2 | MRF24WG wakes up every 200ms to receive buffered messages | | … |  | | 65535 | MRF24WG Wakes up every 65,535 seconds (109 minutes) to receive buffered messages | |
| dtimInterval | Only used if *useDtim* is true. The DTIM period indicates how often clients serviced by the access point should check for buffered multicast or broadcast messages awaiting pickup on the access point. The DTIM interval is measured in number of beacon periods. Typically the DTIM period is 2. |
| useDtim | True: check for buffered multicast or broadcast messages on the *dtimInterval*.  False: check for buffered multicast or broadcast messages on the *listenInterval* |

### WF\_PsPollDisable

|  |  |
| --- | --- |
| Prototype | void **WF\_PsPollDisable**(void); |
| Description | Disables PS-Poll mode. |
| Inputs | None |
| Returns | None |

### WF\_Hibernate

|  |  |
| --- | --- |
| Prototype | void **WF\_Hibernate**(void); |
| Description | Calling this function will assert the HIBERNATE line on the MRF24WG, essentially powering it off and holding it in a very low power reset state. All internal state is lost. This function should be used when the application will not need WiFi communications for a period of time.  After calling this function, the only way to take the MRF24WG out of reset is to call or power cycle the device. |
| Inputs | None |
| Returns | None |

### WF\_PowerStateGet

|  |  |
| --- | --- |
| Prototype | void **WF\_PoweStateGet**(uint8\_t \*p\_powerState); |
| Description | Retrieves the MF24WG power state. |
| Inputs | p\_powerState – pointer to where the power state will be written  Values:  WF\_PS\_HIBERNATE  WF\_PS\_PS\_POLL\_DTIM\_ENABLED  WF\_PS\_PS\_POLL\_DTIM\_DISABLED  WF\_PS\_OFF |
| Returns | None |

## WiFi Maximum Power Functions

### WF\_TxPowerFactoryMaxGet

|  |  |
| --- | --- |
| Prototype | void **WF\_TxPowerGetFactoryMax**(uint8\_t \*p\_maxPower); |
| Description | Retrieves the MF24WG factory max Tx power state. This is the highest Tx power the MRF24WG will transmit at. |
| Inputs | p\_maxPower – pointer to where factory max tx power will be written (in dBm). |
| Returns | None |

### WF\_TxPowerMaxSet

|  |  |
| --- | --- |
| Prototype | void **WF\_TxPowerMaxSet**(uint8\_t maxTxPower); |
| Description | Sets the maximum Tx power in dBm. |
| Inputs | maxTxPower – 0 to N (where N is the factor maximum) |
| Returns | None |

## WiFi Status Functions

### WF\_DeviceInfoGet

|  |  |
| --- | --- |
| Prototype | void **WF\_DeviceInfoGet**(t\_deviceInfo \*p\_info); |
| Description | Retrieves information about the MRF24WG. |
| Inputs | p\_info – pointer to where info will be written. See Section 6.9.1.1. |
| Returns | None |

#### t\_deviceInfo

typedef struct mrf24wgDeviceInfo

{

uint8\_t deviceType;

uint8\_t romVersion;

uint8\_t patchVersion;

} t\_**deviceInfo**;

| **Fields** | **Description** |
| --- | --- |
| deviceType | Should always be equal to WF\_MRF24WG\_DEVICE |
| romVersion | ROM version number |
| patchVersion | Patch version number |

### WF\_MacStatsGet

|  |  |
| --- | --- |
| Prototype | void **WF\_MacStatsGet**(t\_macStats \*p\_stats); |
| Description | Retrieves MAC information from the MRF24WG |
| Inputs | p\_stats – pointer to where MAC stats will be written. See Section 6.9.2.1 |
| Returns | None |

#### t\_macStats

typedef struct WFMacStatsStruct

{

uint32\_t MibWEPExcludeCtr;

uint32\_t MibTxBytesCtr;

uint32\_t MibTxMulticastCtr;

uint32\_t MibTxFailedCtr;

uint32\_t MibTxRtryCtr;

uint32\_t MibTxMultRtryCtr;

uint32\_t MibTxSuccessCtr;

uint32\_t MibRxDupCtr;

uint32\_t MibRxCtsSuccCtr;

uint32\_t MibRxCtsFailCtr;

uint32\_t MibRxAckFailCtr;

uint32\_t MibRxBytesCtr;

uint32\_t MibRxFragCtr;

uint32\_t MibRxMultCtr;

uint32\_t MibRxFCSErrCtr;

uint32\_t MibRxWEPUndecryptCtr;

uint32\_t MibRxFragAgedCtr;

uint32\_t MibRxMICFailureCtr;

} t\_macStats;

| **Fields** | **Description** |
| --- | --- |
| MibWEPExcludeCtr | Number of frames received with the Protected Frame subfield of the Frame Control field set to zero and the value of dot11ExcludeUnencrypted causes that frame to be discarded |
| MibTxBytesCtr | Total number of Tx bytes that have been transmitted |
| MibTxMulticastCtr | Number of frames successfully transmitted that had the multicast bit set in the destination MAC address. |
| MibTxFailedCtr | Number of Tx frames that failed due to the number of transmits exceeding the retry count |
| MibTxRtryCtr | Number of times a transmitted frame needed to be retried |
| MibTxMultRtryCtr | Number of times a frame was successfully transmitted after more than one retransmission. |
| MibTxSuccessCtr | Number of Tx frames successfully transmitted |
| MibRxDupCtr | Number of frames received where the Sequence Control field indicates a duplicate. |
| MibRxCtsSuccCtr | Number of CTS frames received in response to an RTS frame. |
| MibRxCtsFailCtr | Number of times an RTS frame was not received in response to a CTS frame |
| MibRxAckFailCtr | Number of times an Ack was not received in response to a Tx frame |
| MibRxBytesCtr | Total number of Rx bytes received |
| MibRxFragCtr | Number of successful received frames (management or data) |
| MibRxMultCtr | Number of frames received with the multicast bit set in the destination MAC address |
| MibRxFCSErrCtr | Number of frames received with an invalid Frame Checksum (FCS) |
| MibRxWEPUndecryptCtr | Number of frames received where the Protected Frame subfield of the Frame Control Field is set to one and the WEPOn value for the key mapped to the transmitter’s MAC address indicates the frame should not have been encrypted. |
| MibRxFragAgedCtr | Number of times that fragments ‘aged out’, or were not received in the allowable time |
| MibRxMICFailureCtr | Number of MIC failures that have occurred |

## WiFi Multicast Filter Functions

The MRF24WG supports two types of multicast filters, only one of which can be used during an application. This is controlled by the #defines described in Section 5.6. Once a multicast filter is enabled, only those messages that match the filter are forwarded to the host.

A summary of the different multicast filter types is shown below:

|  |  |
| --- | --- |
| **Hardware Multicast Filtering** | **Software Multicast Filtering** |
| 2 filters supported | 16 filters supported |
| Uses hardware, so very efficient | Uses software, so less efficient |
| Can only filter on the entire MAC address | Can filter on a partial MAC address |

### WF\_HwMulticastFilterSet

|  |  |
| --- | --- |
| Prototype | void **WF\_HwMulticastFilterSet**(uint8\_t multicastFilterId,  uint8\_t multicastAddress[WF\_MAC\_ADDRESS\_LENGTH]); |
| Description | Configures one of the two hardware multicast filters. |
| Inputs | |  |  | | --- | --- | | MulticastFilterId – | Will be either WF\_MULTICAST\_FILTER\_1 or WF\_MULTICAST\_FILTER\_2 | | multicastAddress – | 6-byte multicast address to filter on. Setting all 6 bytes to 0xFF will disable the hardware multicast filter. | |
| Returns | None |

### WF\_SwMulticastFilterSet

|  |  |
| --- | --- |
| Prototype | void **WF\_SwMulticastFilterSet**(t\_swMulticastConfig \*p\_config); |
| Description | Configures one of the sixteen software multicast filters. By default all multicast filters are disabled. |
| Inputs | p\_config – pointer to software multicast configuration. See Section 6.10.2.1. |
| Returns | None |
| Example | The goal is filter on Multicast Address of 01:00:5e:xx:xx:xx where xx are don't care bytes.  p\_config->filterId = WF\_MULTICAST\_FILTER\_1;  p\_config->macBytes[0] = 0x01; // need exact match  p\_config->macBytes[1] = 0x00; // need exact match  p\_config->macBytes[2] = 0x5e; // need exact match  p\_config->macBytes[3] = 0xff; // don’t care byte  p\_config->macBytes[4] = 0xff; // don’t care byte  p\_config->macBytes[5] = 0xff; // don’t care byte  p\_config->macBitMask = 0x38 // don't care on bytes 3,4,5 |

#### t\_swMulticastConfig

typedef struct swMulticastConfigStruct

{

uint8\_t filterId;

uint8\_t action;

uint8\_t macAddress[WF\_MAC\_ADDRESS\_LENGTH];

uint8\_t macBitMask;

} t\_swMulticastConfig;

| **Fields** | **Description** |
| --- | --- |
| filterId | Software filter ID.  Range: WF\_MULTICAST\_FILTER\_1 thru WF\_MULTICAST\_FILTER\_16 |
| action | Choices are:   |  |  | | --- | --- | | WF\_MULTICAST\_DISABLE\_ALL | The Multicast Filter discards all received  multicast messages -- they will not be forwarded to the Host MCU. The remaining fields in this structure are ignored. | | WF\_MULTICAST\_ENABLE\_ALL | The Multicast Filter forwards all received multicast messages to the Host MCU. The remaining fields in this structure are ignored. | | WF\_MULTICAST\_USE\_FILTERS | The MAC filter will be used and the remaining fields in this structure configure which Multicast messages are forwarded to the Host MCU. | |
| macBytes | Array containing the MAC address to filter on (using the destination address of each incoming 802.11 frame). Specific bytes within the MAC address can be designated as “don’t care” bytes. See macBitMask.  This field in only used if *action* = WF\_MULTICAST\_USE\_FILTERS. |
| macBitMask | A byte where bits 5:0 correspond to macBytes[5:0]. If the bit is zero then the corresponding MAC byte must be an exact match for the frame to be forwarded to the Host PIC. If the bit is one then the corresponding MAC byte is a “don’t care” and not used in the Multicast filtering process.  This field in only used if *action* = WF\_MULTICAST\_USE\_FILTERS. |

## WiFi Packet Tx Functions

This section contains functions for transmitting an 802.3 Ethernet packet to the wireless network. There are three steps to transmitting a packet, involving three functions that must be called in the following order:

|  |  |
| --- | --- |
|  | Allocates memory on the MRF24WG to hold the packet |
|  | Copies Tx packet from Host memory to MRF24WG memory (via SPI) |
|  | Signals the MRF24WG to transmit the packet onto the wireless network |

### WF\_TxPacketAllocate

|  |  |
| --- | --- |
| Prototype | bool **WF\_TxPacketAllocate**(uint16\_t packetSize); |
| Description | Allocates memory on the MRF24WG to hold a Tx packet. The MRF24WG has limited buffer space, so it is possible for this function to return false occasionally, especially when heavy data traffic with large packets is taking place. |
| Inputs | packetSize – number of bytes in the Tx packet, including Ethernet header |
| Returns | True if successful, else False |

### WF\_TxPacketCopy

|  |  |
| --- | --- |
| Prototype | void **WF\_TxPacketCopy**(uint8\_t \*p\_buf, uint16\_t length); |
| Description | Transfers packet data from Host memory to the MRF24WG memory block created in . The entire packet can be transferred in a single function call. Or, if needed, multiple calls can be made to this function to transfer the packet in chunks. For example, presume a TCP/IP stack stores a packet in a chain of linked buffers. The transmit code would look like:  P\_buf = first buffer in chain  WHILE not at last buffer  WF\_**TxPacketCopy**(p\_buf, length\_of\_this\_buffer);  P\_buf = next buffer in chain  ENDWHILE |
| Inputs | p\_buf – pointer to buffer being transferred  length – number of bytes in p\_buf |
| Returns | None |

### WF\_TxPacketTransmit

|  |  |
| --- | --- |
| Prototype | void **WF\_TxPacketTransmit**(uint16\_t packetSize); |
| Description | Signals the MRF24WG to transmit the packet. |
| Inputs | packetSize – number of bytes in the Tx packet, including Ethernet header. This must be the same value as was used in . |
| Returns | None |

## WiFi Packet Rx Functions

An application is notified that an Rx packet has arrived via the callback function . There are three steps to processing an Rx packet, involving three functions that must be called in the following order:

|  |  |
| --- | --- |
|  | returns the length of the Rx packet |
|  | reads the packet by transferring data from the MRF24WG to host memory |
|  | informs the MRF24WG that it can deallocate memory that was used for the Rx packet |

### WF\_RxPacketLengthGet

|  |  |
| --- | --- |
| Prototype | uint16\_t **WF\_RxPacketLengthGet**(void); |
| Description | Returns the length, in bytes, of the currently received Rx packet, including Ethernet header |
| Inputs | None |
| Returns | Length of received packet. If an error occurred, 0 will be returned. |

### WF\_RxPacketCopy

|  |  |
| --- | --- |
| Prototype | void **WF\_RxPacketCopy**(uint8\_t \*p\_buf, uint16\_t length); |
| Description | Copies an Rx packet from MRF24WG memory to Host memory. The entire packet can be read in a single function call, or, this function can be called multiple times to retrieve the entire Rx packet. For example, if an application had a chain of variable-sized buffers to hold an Rx packet the code would look like:  p\_buf = first Rx buffer in chain  while(1)  {  **WF\_RxPacketCopy**(p\_buf, size\_of\_this\_rx\_buffer);  p\_buf = next buffer in chain  IF done reading entire packet  break;  ENDIF  } |
| Inputs | p\_buf – location where Rx bytes are written to  length – number of bytes to read |
| Returns | None |

### WF\_RxPacketDeallocate

|  |  |
| --- | --- |
| Prototype | void **WF\_RxPacketDeallocate**(void); |
| Description | After the host has copied the entire Rx packet from the MRF24WG into host memory, this function must be called to free the MRF24WG memory that was used to hold the Rx packet. |
| Inputs | None |
| Returns | None |

## WiFi Miscellaneous Functions

### WF\_MacAddressGet

|  |  |
| --- | --- |
| Prototype | void **WF\_MacAddressGet**(uint8\_t \*p\_macAddress); |
| Description | Retrieves the MRF24WG MAC address. |
| Inputs | p\_macAddress – pointer where MAC address will be copied. MAC address will always be 6 bytes. |
| Returns | None |

### WF\_MacAddressSet

|  |  |
| --- | --- |
| Prototype | void **WF\_MacAddressSet**(uint8\_t \*p\_macAddress); |
| Description | Directs the MRF24WG to use the input MAC address instead of its factory-default MAC address. This function does not overwrite the factory default – it simply tells the MRF24WG to use a different MAC. |
| Inputs | p\_macAddress – pointer to MAC address |
| Returns | None |

## WiFi Advanced Functions

Functions in this section are not needed by most applications, as the MRF24WG defaults are generally sufficient. Advanced users may need one or more of the functions in this section.

### WF\_ScanContextSet

|  |  |
| --- | --- |
| Prototype | void **WF\_ScanContextSet**(t\_scanContext \*p\_context); |
| Description | Sets parameters related to WiFi scanning. |
| Inputs | p\_context – see Section 6.14.1.1 |
| Returns | None |

#### t\_scanContext

typedef struct scanContext

{

uint8\_t scanType;

uint8\_t scanCount;

uint16\_t minChannelTime;

uint16\_t maxChannelTime;

uint16\_t probeDelay;

} t\_scanContext;

| **Fields** | **Description** |
| --- | --- |
| scanType | 802.11 allows for active scanning, where the device sends out a broadcast probe request seeking an access point. Also allowed is passive scanning where the device only listens to beacons being broadcast from access points.  Options: WF\_ACTIVE\_SCAN or WF\_PASSIVE\_SCAN  Default: WF\_ACTIVE\_SCAN |
| scanCount | The number of times to scan a channel while attempting to find a particular access point.  Default: 1 |
| minChannelTime | The minimum time (in milliseconds) the MRF24WG will wait for a probe response after sending a probe request. If no probe responses are received in *minChannelTime* then the MRF24WG will go on to the next channel, if any are left to scan, or quit.  Default: 200ms |
| maxChannelTime | If a probe response is received within *minChannelTime* then the MRF24WG will continue to collect any additional probe responses up to *maxChannelTime* before going to the next channel in the *channelList*. Units are in milliseconds.  Default: 400ms |
| probeDelay | The number of microseconds to delay before transmitting a probe request following the channel change during scanning.  Default: 20uS |

### WF\_AdhocContextSet

|  |  |
| --- | --- |
| Prototype | void **WF\_AdhocContextSet**(t\_adHocNetworkContext \*p\_context); |
| Description | Sets parameters related to AdHoc Networks. |
| Inputs | p\_context – see Section 6.14.2.1 |
| Returns | None |

#### t\_adHocNetworkContext

typedef struct adHocNetworkContext

{

bool hiddenSsid;

uint16\_t beaconPeriod;

uint8\_t mode;

} t\_adHocNetworkContext;

| **Fields** | **Description** |
| --- | --- |
| hiddenSsid | The SSID in an AdHoc network can be hidden.  Options: True or False  Default: False |
| beaconPeriod | Beacon period, in ms.  Default: 100ms |
| mode | Defines how to start the AdHoc network.  Options:   |  |  | | --- | --- | | WF\_ADHOC\_CONNECT\_THEN\_START | First attempt to connect to an AdHoc network with the specified SSID. If it can’t be found, then start an AdHoc network. | | WF\_ADHOC\_CONNECT\_ONLY | Only connect to an existing AdHoc network if it can be found. Do not start an AdHoc network. | | WF\_ADHOC\_START\_ONLY | Only start an AdHoc network; do not join any existing AdHocnetworks. |   Default: WF\_ADHOC\_CONNECT\_THEN\_START |

### WF\_TxModeSet

|  |  |
| --- | --- |
| Prototype | void **WF\_TxModeSet**(uint8\_t mode); |
| Description | Configure the MRF24WG transmit mode |
| Inputs | mode -- Select from one of the following:   |  |  | | --- | --- | | WF\_TXMODE\_G\_RATES | Use all ‘g’ rates | | WF\_TXMODE\_B\_RATES | Use only ‘b’ rates | | WF\_TXMODE\_LEGACY\_RATES | Use only rates as originally used by MRF24WB |   Default: WF\_TXMODE\_G\_RATES |
| Returns | None |

### WF\_BssidSet

|  |  |
| --- | --- |
| Prototype | void **WF\_BssidSet**(uint8\_t \*p\_bssid); |
| Description | Basic Service Set Identifier, always 6 bytes. This is the 48-bit MAC of the SSID. It is an optional field that can be used to specify a specific SSID if more than one AP exists with the same SSID. This field can also be used in lieu of the SSID. Set each byte to 0xFF if BSSID is not going to be used. |
| Inputs | p\_bssid – pointer to 6-byte BSSID  Default: BSSID not used (all FF’s) |
| Returns | None |

### WF\_RssiSet

|  |  |
| --- | --- |
| Prototype | void **WF\_RtsThresholdSet**(uint16\_t rstThreshold); |
| Description | Specifies RSSI restrictions when connecting. This field is only used if:   1. Neither an SSID or BSSID has been configured   or   1. An SSID is defined and multiple AP’s are discovered with the same SSID |
| Inputs | rssi – RSSI value   |  |  | | --- | --- | | **RSSI** | **Description** | | 0 | Connect to the first network found | | 1-254 | Only connect to a network if the RSSI is greater than or equal to the specified value. | | 255 | Connect to the highest RSSI found (default) |   Default: 255 |
| Returns | None |

### WF\_RtsThresholdSet

|  |  |
| --- | --- |
| Prototype | void **WF\_RtsThresholdSet**(uint16\_t rstThreshold); |
| Description | Sets the RTS/CTS packet size threshold for when RTS/CTS frame will be sent. |
| Inputs | rtsThreshold – 0 to 2347  Default: 2347 |
| Returns | None |

### WF\_LinkDownThresholdSet

|  |  |
| --- | --- |
| Prototype | void **WF\_LinkDownThresholdSet**(uint8\_t threshold); |
| Description | Configures the number of WiFi transmission failures that can occur before  the MRF24WG reports the connection is lost.  There are two primary ways the MRF24WG can determine that it has lost the link with an AP:  1) beacon timeout  2) receive deauth message from AP  The deauth message, however, is not ack'd by the MRF24WG, so it is not guaranteed that it will be received by the MRF24WG. In this case, the MRF24WG will only  know it has lost the link because none of its future messages will be ack'd by  the AP. This function allows the host to configure a 'missed ack' count, where,  if the MRF24WG misses N consecutive ack's from the AP it generates an event that  the connection is lost.  The default threshold is 0, meaning that the MRF24WG ignores missed ack's, and  will simply keep trying to communicate with the AP. It is important to note that  in normal operations, ack's from the AP are frequently missed, and retries are  quite common, so, if using this function, do not set the threshold to low because  then erroneous connection lost events will be generated. It is recommended  that a threshold of 40 be used if this function is to be utilized. |
| Inputs | Threshold – 0 disables this feature; MRF24WG does not tracked missed ack’s  1:255 after this number of missed ack’s, connection declared lost  Default: 0 |
| Returns | None |

# Event Data

Some of the events described in Section 4.5.1 use the *eventData* parameter in a more complex fashion. This section describes the *eventData* usage for those events.

## WF\_EVENT\_INITIALIZATION

|  |  |
| --- | --- |
| **eventData** | **Description** |
| WF\_INIT\_SUCCESSFUL | Initialization completed successfully |
| WF\_INIT\_ERROR\_SPI\_NOT\_CONNECTED | Initialization failed – appears SPI interface to MRF24WG is incorrect |
| WF\_INIT\_ERROR\_RESET\_TIMEOUT | Initialization failed – timed out waiting for MRF24WG to reset |
| WF\_INIT\_ERROR\_INIT\_TIMEOUT | Initialization failed – timed out waiting for MRF24WG to complete its initialization |

## WF\_EVENT\_CONNECTION\_TEMPORARILY\_LOST

*eventData* is defined as:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Deauth | Disassoc | Event Data | | | | | |

Bits 31:8 are not used

If *Deauth* = 1 or *Disassoc* = 1 then Event Data is:

Table : Deauth or Dissaoc Event Data

|  |
| --- |
| DD\_UNSPECIFIED |
| DD\_PREV\_AUTH\_NOT\_VALID |
| DD\_DISASSOC\_DUE\_TO\_INACTIVITY |
| DD\_DISASSOC\_AP\_BUSY |
| DD\_CLASS2\_FRAME\_FROM\_NONAUTH\_STA |
| DD\_CLASS3\_FRAME\_FROM\_NONASSOC\_STA |
| DD\_DISASSOC\_STA\_HAS\_LEFT |
| DD\_STA\_REQ\_ASSOC\_WITHOUT\_AUTH |
| DD\_INVALID\_IE |
| DD\_MIC\_FAILURE |
| DD\_4WAY\_HANDSHAKE\_TIMEOUT |
| DD\_GROUP\_KEY\_HANDSHAKE\_TIMEOUT |
| DD\_IE\_DIFFERENT |
| DD\_INVALID\_GROUP\_CIPHER |
| DD\_INVALID\_PAIRWISE\_CIPHER |
| DD\_INVALID\_AKMP |
| DD\_UNSUPP\_RSN\_VERSION |
| DD\_INVALID\_RSN\_IE\_CAP |
| DD\_IEEE8021X\_FAILED |
| DD\_CIPHER\_SUITE\_REJECTED |

If both *Deauth* and *Disassoc* equal 0 then Event Data is:

Table : Connection Lost Event Data

|  |
| --- |
| CL\_ASSOCIATION\_FAILURE |
| CL\_BEACON\_TIMEOUT |
| CL\_DEAUTH\_RECEIVED |
| CL\_DISASSOCIATE\_RECEIVED |
| CL\_TKIP\_MIC\_FAILURE |
| CF\_LINK\_DOWN |

## WF\_EVENT\_CONNECTION\_FAILED

*eventData* is defined as:

|  |  |
| --- | --- |
| 15:8 | 7:0 |
| Status | Reason |

*Status* is:

Table : Connection Failures

|  |
| --- |
| CF\_JOIN\_FAILURE |
| CF\_AUTHENTICATION\_FAILURE |
| CF\_ASSOCIATION\_FAILURE |
| CF\_WEP\_HANDSHAKE\_FAILURE |
| CF\_PSK\_CALCULATION\_FAILURE |
| CF\_PSK\_HANDSHAKE\_FAILURE |
| CF\_ADHOC\_JOIN\_FAILURE |
| CF\_SECURITY\_MISMATCH\_FAILURE |
| CF\_NO\_SUITABLE\_AP\_FOUND\_FAILURE |
| CF\_RETRY\_FOREVER\_NOT\_SUPPORTED\_FAILURE |
| CF\_LINK\_LOST |
| CF\_TKIP\_MIC\_FAILURE |
| CF\_RSN\_MIXED\_MODE\_NOT\_SUPPORTED |
| CF\_RECV\_DEAUTH |
| CF\_RECV\_DISASSOC |
| CF\_WPS\_FAILURE |
| CF\_P2P\_FAILURE |
| CF\_LINK\_DOWN |

If *Status* = CF\_RECV\_DEAUTH or *Status* = CF\_RECV\_DISASSOC:

then *Reason* is one of the values from Table 6

If *Status* = CF\_AUTHENTICATION\_FAILURE or *Status* = CF\_ASSOCIATION\_FAILURE:

Then *Reason* is:

Table : Authentication and Association Failures

|  |
| --- |
| SC\_UNSPECIFIED\_FAILURE |
| SC\_CAPS\_UNSUPPORTED |
| SC\_REASSOC\_NO\_ASSOC |
| SC\_ASSOC\_DENIED\_UNSPEC |
| SC\_NOT\_SUPPORTED\_AUTH\_ALG |
| SC\_UNKNOWN\_AUTH\_TRANSACTION |
| SC\_CHALLENGE\_FAIL |
| SC\_AUTH\_TIMEOUT |
| SC\_AP\_UNABLE\_TO\_HANDLE\_NEW\_STA |
| SC\_ASSOC\_DENIED\_RATES |
| SC\_ASSOC\_DENIED\_NOSHORTPREAMBLE |
| SC\_ASSOC\_DENIED\_NOPBCC |
| SC\_ASSOC\_DENIED\_NOAGILITY |
| SC\_ASSOC\_DENIED\_NOSHORTTIME |
| SC\_ASSOC\_DENIED\_NODSSSOFDM |
| SC\_NOT\_VALID\_IE |
| SC\_NOT\_VALID\_GROUPCIPHER |
| WF\_NOT\_VALID\_PAIRWISE\_CIPHER |
| WF\_NOT\_VALID\_AKMP |
| WF\_UNSUPPORTED\_RSN\_VERSION |
| WF\_INVALID\_RSN\_IE\_CAP |
| WF\_CIPHER\_SUITE\_REJECTED |
| WF\_TIMEOUT |

If *Status* = WF\_WPS\_FAILURE

Then *Reason* is defined as:

|  |  |
| --- | --- |
| 7:4 | 3:0 |
| wpsState | wpsConfigErr |

Table : wpsState

|  |
| --- |
| EAP\_EAPOL\_START |
| EAP\_EAP\_REQ\_IDENTITY |
| EAP\_EAP\_RSP\_IDENTITY |
| EAP\_EAP\_WPS\_START |
| EAP\_EAP\_RSP\_M1 |
| EAP\_EAP\_REQ\_M2 |
| EAP\_EAP\_RSP\_M3 |
| EAP\_EAP\_REQ\_M4 |
| EAP\_EAP\_RSP\_M5 |
| EAP\_EAP\_REQ\_M6 |
| EAP\_EAP\_RSP\_M7 |
| EAP\_EAP\_REQ\_M8 |
| EAP\_EAP\_RSP\_DONE |
| EAP\_EAP\_FAILURE |

Table : wpsConfigErr

|  |
| --- |
| WPS\_NOERR |
| WPS\_SESSION\_OVERLAPPED |
| WPS\_DECRYPT\_CRC\_FAILURE |
| WPS\_24G\_NOT\_SUPPORTED |
| WPS\_RETRY\_FAILURE |
| WPS\_INVALID\_MSG |
| WPS\_AUTH\_FAILURE |
| WPS\_ASSOC\_FAILURE |
| WPS\_MSG\_TIMEOUT |
| WPS\_SESSION\_TIMEOUT |
| WPS\_DEVPASSWD\_AUTH\_FAILURE |
| WPS\_NO\_CONN\_TOREG |
| WPS\_MULTI\_PBC\_DETECTED |
| WPS\_EAP\_FAILURE |
| WPS\_DEV\_BUSY |
| WPS\_SETUP\_LOCKED |

If *Status* = CF\_P2P\_FAILURE

Then *Reason* is defined as:

|  |  |
| --- | --- |
| 7:4 | 3:0 |
| p2pState | p2pErrro |

Table : p2pState

|  |
| --- |
| P2P\_IDLE |
| P2P\_SCAN |
| P2P\_LISTEN |
| P2P\_FIND |
| P2P\_START\_FORMATION |
| P2P\_NEG\_REQ\_DONE |
| P2P\_WAIT\_NEG\_REQ\_DONE |
| P2P\_WAIT\_FORMATION\_DONE |
| P2P\_INVITE |
| P2P\_PROVISION |
| P2P\_CLIENT |

Table : p2pError

|  |
| --- |
| WFD\_SUCCESS |
| WFD\_INFO\_CURRENTLY\_UNAVAILABLE |
| WFD\_INCOMPATIBLE\_PARAMS |
| WFD\_LIMIT\_REACHED |
| WFD\_INVALID\_PARAMS |
| WFD\_UNABLE\_TO\_ACCOMMODATE |
| WFD\_PREV\_PROTOCOL\_ERROR |
| WFD\_NO\_COMMON\_CHANNELS |
| WFD\_UNKNOWN\_GROUP |
| WFD\_INCOMPATIBLE\_PROV\_METHOD |
| WFD\_REJECTED\_BY\_USER |
| WFD\_NO\_MEM |
| WFD\_INVALID\_ACTION |
| WFD\_TX\_FAILURE |
| WFD\_TIME\_OUT |

## WF\_EVENT\_CONNECTION\_PERMANENTLY\_LOST

*eventData* is defined as:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Deauth | Disassoc | Event Data | | | | | |

Bits 31:8 are not used

If *Deauth* = 1 or *Disassoc* = 1 then Event Data is:

See Table 6

If both *Deauth* and *Disassoc* equal 0 then Event Data is:

See Table 7

# Connection Examples

This section contains code fragment examples of common MRF24WG connection scenarios.

## Infrastructure Connection, No Security

This code fragment connects with an existing Access Point in infrastructure mode with open security.

uint8\_t ssid[] = “MyAccessPointSsid”;

uint8\_t channelList[] = {1,6,11};

WF\_RegionalDomainSet(WF\_DOMAIN\_FCC);

WF\_SsidSet(ssid, strlen(ssid));

WF\_NetworkTypeSet(WF\_NETWORK\_TYPE\_INFRASTRUCTURE);

WF\_ChannelListSet(channelList, sizeof(channelList));

WF\_ReconnectModeSet(WF\_RETRY\_FOREVER, WF\_ATTEMPT\_TO\_RECONNECT, 20,

WF\_ATTEMPT\_TO\_RECONNECT);

WF\_SecurityOpenSet();

// start connection process (WF\_Task must be called repeatedly after this call)

WF\_Connect();

// Successful connection will be signaled in WFProcessEvent()

## AdHoc Connection, No Security

This code fragment causes the MRF24WG to first check for an AdHoc network, and it cannot find one after three tries, it will create it’s own AdHoc network.

uint8\_t ssid[] = "MyAdHocSsid";

uint8\_t channelList[] = {1};

t\_adHocNetworkContext adhocContext;

WF\_RegionalDomainSet(WF\_DOMAIN\_FCC);

WF\_SsidSet(ssid, strlen(ssid));

// set network type and configure AdHoc context

WF\_NetworkTypeSet(WF\_NETWORK\_TYPE\_ADHOC);

adhocContext.hiddenSsid = false;

adhocContext.mode = WF\_ADHOC\_CONNECT\_THEN\_START;

adhocContext.beaconPeriod = WF\_DEFAULT\_ADHOC\_BEACON\_PERIOD;

WF\_AdhocContextSet(&adhocContext);

WF\_ChannelListSet(channelList, sizeof(channelList));

WF\_ReconnectModeSet(3, WF\_ATTEMPT\_TO\_RECONNECT, 20, WF\_ATTEMPT\_TO\_RECONNECT);

WF\_SecurityOpenSet();

// start connection process (WF\_Task must be called repeatedly after this call)

WF\_Connect();

// Successful connection will be signaled in WFProcessEvent()

**Note**: that the first parameter of WF\_ReconnectModeSet() is 3. Do not use WF\_RETRY\_FOREVER for this parameter. The goal is for the MRF24WG to try three times to find the AdHoc network “MyAdHocSsid”, and then start the AdHoc network itself. If retries were set to forever, and the AdHoc network did not already exist, then the MRF24WG would keep looking for it and never start the AdHoc network itself.

## P2P (WiFi Direct) Connection, Push Button

This code fragment connects to an existing Access Point in infrastructure mode using WiFi-Direct with a push-button.

// must be in global memory

t\_wpaKeyInfo g\_keyInfo;

// Connection Function

uint8\_t ssid[] = "DIRECT--"; // fixed SSID, do not change

uint8\_t channelList[] = {1,6,11}; // must use this channel list

t\_wpsContext wpsContext;

WF\_RegionalDomainSet(WF\_DOMAIN\_FCC);

WF\_SsidSet(ssid, strlen(ssid));

// set network type and configure AdHoc context

WF\_NetworkTypeSet(WF\_NETWORK\_TYPE\_INFRASTRUCTURE);

WF\_ChannelListSet(channelList, sizeof(channelList));

WF\_ReconnectModeSet(WF\_RETRY\_FOREVER, WF\_ATTEMPT\_TO\_RECONNECT, 20,

WF\_ATTEMPT\_TO\_RECONNECT);

wpsContext.wpsSecurityType = WF\_SECURITY\_WPS\_PUSH\_BUTTON;

wpsContext.wpsPin = NULL; // not used for pushbutton

wpsContext.wpsPinLength = 0; // not used for pushbutton

wpsContext.getPassPhrase = true; // send passphrase back to host for key calc

wpsContext.p\_keyInfo = g\_keyInfo; // passphrase info will be stored here

WF\_SecurityWpsSet(&wpsContext);

// start connection process (WF\_Task must be called repeatedly after this call)

WF\_Connect();

**Note**: In this example, because wpsContext.getPassPhrase is true, the MRF24WG will send back the ASCII passphrase to the host so the host can more efficiently generate the binary key. The expected sequence after the above code block will be:

1. Event WF\_WPS\_EVENT\_KEY\_CALCULATION\_REQUEST occurs
2. WF\_WpsKeyGenerate() must be called
3. Event WF\_EVENT\_CONNECTION\_SUCCESSFUL occurs

## P2P (WiFi Direct) Connection, PIN

This scenario is very similar to except the user must enter a PIN number on the connecting device. Thus, wpsContext.wpsPin and wpsContext.wpsPinLength must be defined. Also, wpsContext.wpsSecurityType must be set to WF\_SECURITY\_WPS\_PIN.

# Appendix

## Memory Requirements

Below are some examples showing the memory requirements for the Universal Driver under various conditions using a Microchip PIC32MX795F512L MCU and the Microchip XC compiler toolchain.

Table : Universal Driver -- No Demo Code

|  |  |  |
| --- | --- | --- |
|  | **No Compiler Optimizations** | **Compiler Optimization of –Os** |
| **All features enabled** | Code: 51 Kbytes  Data: 400 bytes | Code: 28 Kbytes  Data: 400 bytes |
| **Error checking disabled**  **WPA key generation disabled**  **WPS security disabled** | TBD | Code: 17 Kbytes  Data: 344 bytes |

Table : Universal Driver – lwIP Demo

|  |  |  |
| --- | --- | --- |
|  | **No Compiler Optimizations** | **Compiler Optimization of –Os** |
| **All features enabled** | Code: 155.3 Kbytes  Data: 6776 bytes | Code: 73.8 Kbytes  Data: 6784 bytes |
| **Error checking disabled**  **WPA key generation disabled**  **WPS security disabled** | Code: 148.7 Kbytes  Data: 6724 bytes | Code: 71.5 Kbytes  Data: 6732 bytes |