
MiWi™ Wireless Networking Protocol Stack

*Author: David Flowers and Yifeng Yang
Microchip Technology Inc.*

INTRODUCTION

Implementing applications with wireless networking is becoming commonplace. From consumer devices to industrial applications, there is a growing expectation that our devices will have built-in the ability to talk to each other without a hard-wired connection. The challenge is to select the right wireless networking protocol and implement it in a cost-effective manner.

The MiWi™ Wireless Networking Protocol is a simple protocol designed for low data rate, short distance, low-cost networks. Fundamentally based on IEEE 802.15.4™ for wireless personal area networks (WPANs), the MiWi protocol provides an easy-to-use alternative for wireless communication. In particular, it targets smaller applications that have relatively small network sizes, with few hops between nodes, using Microchip's MRF24J40 2.4 GHz transceiver for IEEE 802.15.4 compliant networks.

This application note covers the definition of the MiWi Wireless Networking Protocol Stack and how it works. The example Stack implementation data structures, usage and APIs are covered in this document, as well as resource requirements for this implementation. For completeness, the document also introduces several aspects of wireless networking, as well as key features of IEEE 802.15.4. However, it is assumed that the user is already familiar with the C programming language and IEEE 802.15.4. You are strongly advised to read the specification in detail prior to using the Microchip MiWi Wireless Networking Protocol Stack.

FEATURES

The current implementation of the MiWi protocol has these features:

- Support for the 2.4 GHz spectrum through the MRF24J40 transceiver
- Support for all IEEE 802.15.4 device types
- Portable across PIC16, PIC18, PIC24 and dsPIC33 devices
- RTOS and application independent
- Out-of-box support for the MPLAB® C18 and C30 compilers
- An easy-to-use API

CONSIDERATIONS

A network using the MiWi protocol is capable of having a maximum of 1024 nodes on a network. Each coordinator is only capable of having 127 children, with a maximum of 8 coordinators in a network. Packets can travel a maximum of 4 hops in the network and 2 hops maximum from the PAN coordinator.

If, after reading this application note, you determine that you require a standardized wireless platform, larger network sizes or common marketing logos, please refer to Microchip application note AN965, "The Microchip Stack for the ZigBee™ Protocol" (DS00965). Alternatively, users may wish to consider using the basic MiWi protocol and modifying it to suit their own applications.

For the most up-to-date list of limitations of the Stack, please refer to the Readme file located with the Stack download.

TERMINOLOGY

In describing the MiWi protocol, two specific terms are used throughout that are borrowed from the IEEE standard.

The first term is **cluster**, which refers to a grouping of nodes that form a network. A MiWi protocol cluster can be up to 3 nodes deep and is controlled by a cluster-head. In the current implementation of the MiWi protocol, the cluster-head is always the PAN coordinator. (For further information, see Table 3 on page 2.)

The second term is **socket**, which refers to a virtual connection between two devices. Rather than have an exclusive hard-wired connection between devices, many devices with many types of sockets share a common communications medium and use some common method to associate applications and devices. When a new device or application is added to the network, it requires configuration to talk to other devices or applications. By using sockets, nodes in the network can find communication partners dynamically without having to know any information about them.

MIWI PROTOCOL OVERVIEW

The MiWi protocol is based on the MAC and PHY layers of the IEEE 802.15.4 specification, and is tailored for simple network development in the 2.4 GHz band. The protocol provides the features to find, form and join a network, as well as discovering nodes on the network and route to them. It does not cover any application-specific issues, such as how to select which network to join to, how to decide when a link is broken or how often devices should communicate.

IEEE 802.15.4 PHY and MAC

The MiWi protocol uses IEEE Standard 802.15.4 for the specifications of the Medium Access Layer (MAC) and Physical Layer (PHY). The IEEE 802.15.4 defines three frequency bands of operations: 2.4 GHz, 915 MHz and 868 MHz. Each frequency band offers a fixed number of channels, and has a different maximum bit rate (see Table 1). Note that the actual data throughput will be less than the specified bit rate due to the packet overhead and processing delays.

TABLE 1: AVAILABLE CHANNELS IN IEEE 802.15.4™

Frequency Band	Available Channels (Channel #)	Maximum Data Throughput (kbps)
868 MHz	1 (0)	20
915 MHz	10 (1-10)	40
2.4 GHz	16 (11-26)	250

The maximum length of an IEEE 802.15.4 MAC packet is 127 bytes, including a 16-bit CRC value. The 16-bit CRC value verifies the frame integrity.

In addition, IEEE 802.15.4 optionally uses an Acknowledged data transfer mechanism in the MAC. This method uses a special ACK flag in the packet header. When this flag is set, Acknowledgement to the transmitter by its receiver is required; this ensures that a frame is, in fact, delivered. If the frame is transmitted with an ACK flag set and the Acknowledgement is not received within a certain time-out period, the transmitter will retry the transmission for a fixed number of times before declaring an error.

It is important to note that the reception of an Acknowledgement simply indicates that a frame was properly received by the MAC layer. It does not, however, indicate that the frame was processed correctly. It is possible that the MAC layer of the receiving node received and Acknowledged a frame correctly, but due to the lack of processing resources, a frame might be discarded by upper layers. As a result, the upper layers of the application may require additional Acknowledgement response.

Device Types

IEEE 802.15.4 defines devices based on their overall functionality. There are basically two device types shown in Table 2.

The MiWi protocol defines three types of MiWi protocol devices, based on their functions in the network: PAN Coordinator, Coordinator and End Device. The MiWi Wireless Networking Protocol Stack functionality helps to determine the type of IEEE functionality that the device requires. The MiWi protocol device types and their relationship to IEEE device types are shown in Table 3.

TABLE 2: IEEE 802.15.4™ FUNCTIONAL DEVICE TYPES

Device Type	Services Offered	Typical Power Source	Typical Receiver Idle Configuration
Full Function Device (FFD)	All or Most	Mains	On
Reduced Function Device (RFD)	Limited	Battery	Off

TABLE 3: MIWi™ PROTOCOL DEVICE TYPES

Device Type	IEEE Device Type	Typical Function
PAN Coordinator	FFD	One per network. Forms the network, allocates network addresses, holds binding table.
Coordinator	FFD	Optional. Extends the physical range of the network. Allows more nodes to join the network. May also perform monitoring and/or control functions.
End Device	FFD or RFD	Performs monitoring and/or control functions.

MIWI PROTOCOL NETWORK CONFIGURATIONS

Of the three device types defined in the MiWi protocol, the most central type to networking is the PAN coordinator. The PAN coordinator is the device that starts the network, and selects the channel and the PAN ID of the network. All other devices joining onto the PAN have to obey the instructions of the PAN coordinator.

Star Network Configuration

A star network configuration (Figure 1) consists of one PAN coordinator node and one or more end devices. In a star network, all end devices communicate only with the PAN coordinator. If an end device needs to transfer

data to another end device, it sends its data to the PAN coordinator. The PAN coordinator, in turn, forwards the data to the intended recipient.

Cluster-Tree Network Configuration

In a cluster tree network (Figure 2) there is still only one PAN coordinator; however, other coordinators are allowed to join on to the network. This forms a tree-like structure, where the PAN coordinator is the root of the tree, the coordinators are the branches of the tree and the end devices are the leaves of the tree. In a cluster tree network, all of the messages sent through the network follow the path of the tree structure. Since messages may be routed through more than one node to reach their eventual destination, cluster tree networks are sometimes referred to as multi-hop networks.

FIGURE 1: STAR NETWORK CONFIGURATION

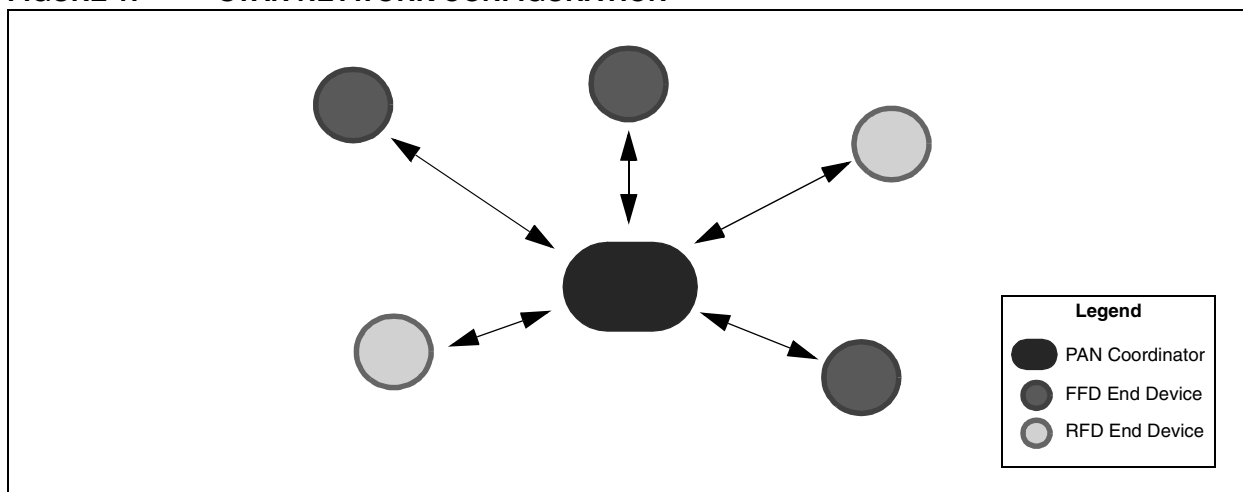
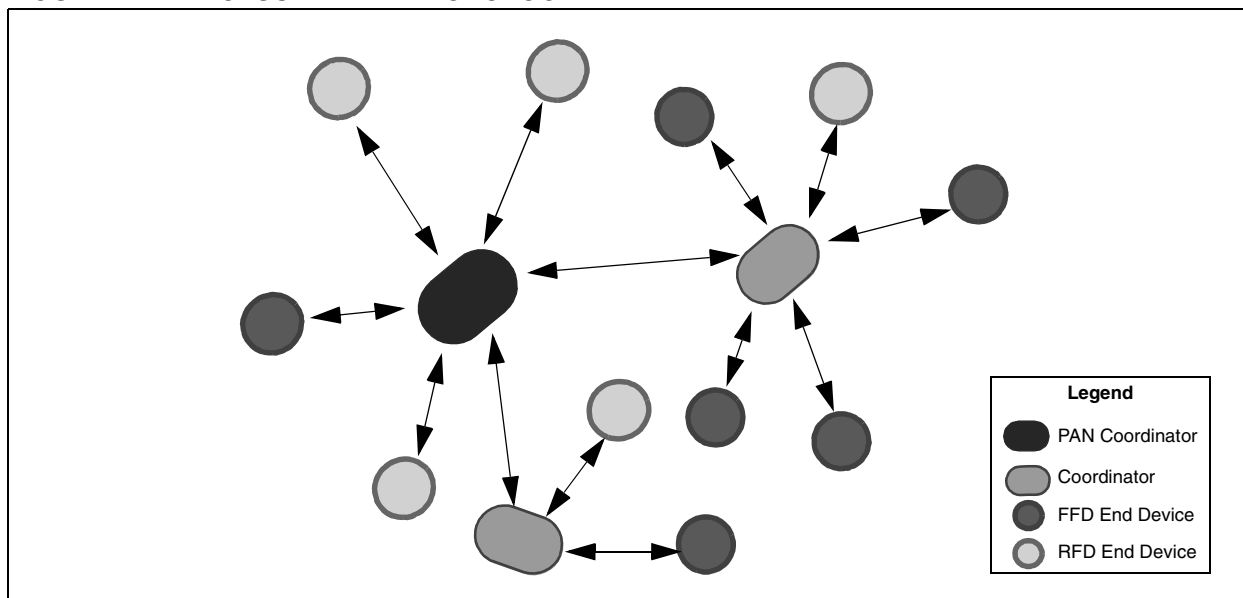


FIGURE 2: CLUSTER TREE TOPOLOGY



Mesh Network Configuration

A mesh network (Figure 3) is similar to a cluster tree configuration, except that FFDs can route messages directly to other FFDs instead of following the tree structure. Messages to RFDs must still go through the RFD's parent node. The advantages of this topology are that message latency can be reduced and reliability is increased. Like cluster tree networks, mesh networks are multi-hop.

Peer-to-Peer (P2P) Configuration

A peer-to-peer configuration is the simplest form of communication, with just one device talking directly to another device. In this configuration, there is no distinction of parent or child, or routing to other nodes.

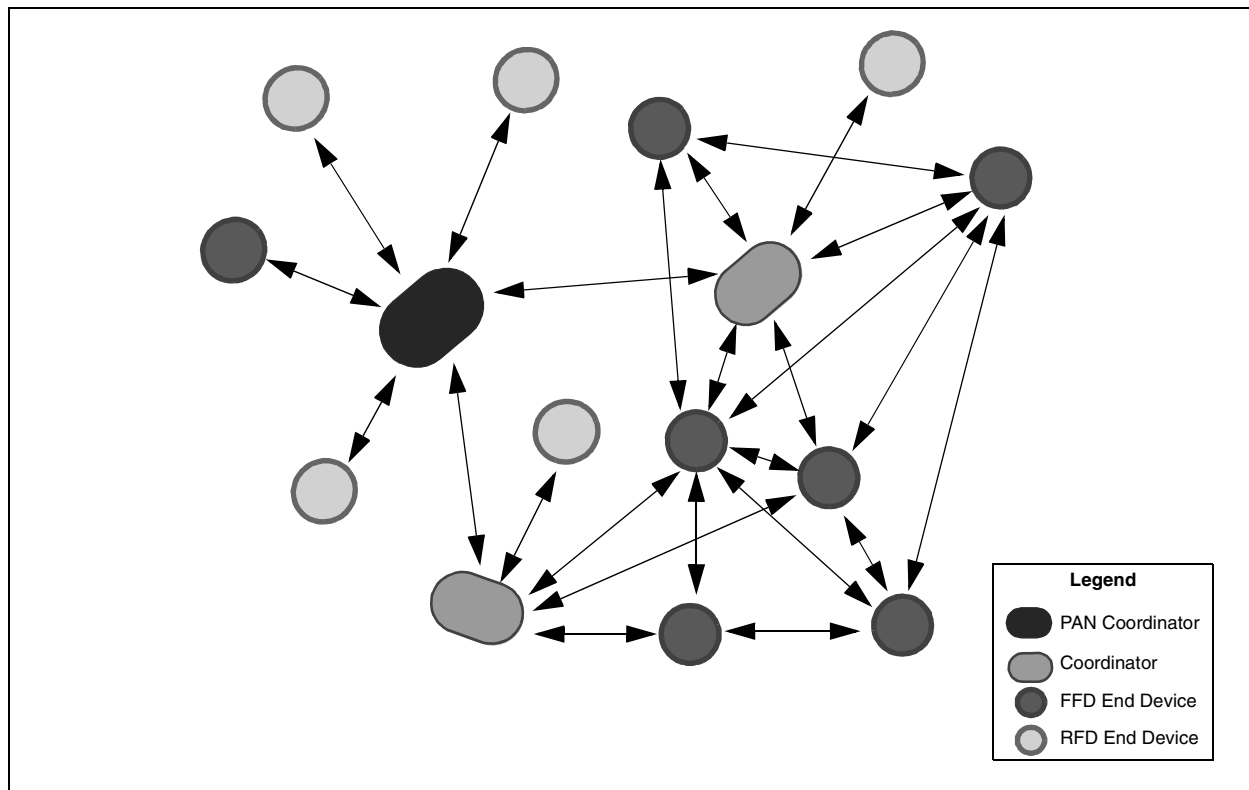
Multi-Access Networks

An IEEE 802.15.4 network is a multi-access network, meaning that all nodes in a network have equal access to the medium of communication. There are two types of multi-access mechanisms: beacon and non-beacon.

In a beacon enabled network, nodes are allowed to transmit in predefined time slots only. The PAN coordinator periodically begins with a superframe, identified as a beacon frame, and all nodes in the network are expected to synchronize to this frame. Each node is assigned a specific slot in the superframe, during which, it is allowed to transmit and receive its data. A superframe may also contain a common slot during which all nodes compete to access the channel.

In a non-beacon enabled network, all nodes in a network are allowed to transmit at any time as long as the channel is Idle. The current version of the Microchip MiWi Wireless Networking Protocol Stack supports only non-beacon networks.

FIGURE 3: MESH NETWORK



ADDRESS ASSIGNMENT

The MiWi protocol uses the addresses provided by IEEE 802.15.4. There are three different addresses defined by the specification:

1. **Extended Organizationally Unique Identifier (EUI):** This is an 8-byte number that is globally unique. Every device ever shipped in the world, using the IEEE 802.15.4 specification, should have a unique EUI address. The upper 3 bytes of the EUI are purchased from IEEE (see link in the reference section for the site to buy them). The lower 5 bytes of the EUI are available for the user, as they see fit, as long as they are globally unique.
2. **PAN Identifier (PANID):** The PANID is a 16-bit address that defines a group of nodes. All of the nodes in the PAN share a common PANID. A device assumes the PANID for a network when it selects to join that PAN.
3. **Short Address:** Also known as the device address, this is a 16-bit (2-byte) address that is assigned to a device by its parent. This short address is unique within a PAN and is used for addressing and messaging within the network. IEEE specifies that the PAN coordinator always has an address of 0000h. The address allocation is up to the PAN coordinator from that point forward.

The MiWi protocol uses the 16 available bits in the short address to help with routing and exchanging node information. The bit fields within the address are shown in Figure 4.

The Parent's Number field (bits 10-8) is unique for each coordinator on the network, including the PAN coordinator. As the Parent's Number field is only 3 bits long, this limits the number of coordinators in a network to 8.

The Child's Number field (bits 6-0) of any coordinator on the network will be 00h. This indicates that they are operating as a coordinator. Other values for this field are determined by the type of device (FFD or RFD), as well its function within the PAN. Figure 5 gives a general idea of how short addresses are determined.

The RxOffWhenIdle field (bit 7) is the inverse of the IEEE 802.15.4 defined property of RxOnWhenIdle. When this bit is set, it indicates that this device will turn off its transceiver when it is Idle and will be unable to receive packets. Any device, other than this device's parent, should route any packets that have this bit set to the device's parent. The target device's parent will buffer the message for the child until it wakes up and requests the data. If this bit is not set in the device's address, then this device is always capable of receiving packets.

Bits 15 through 11 are always '0' in this implementation.

FIGURE 4: BIT FIELD ARRANGEMENT FOR THE MIWi™ PROTOCOL SHORT ADDRESS

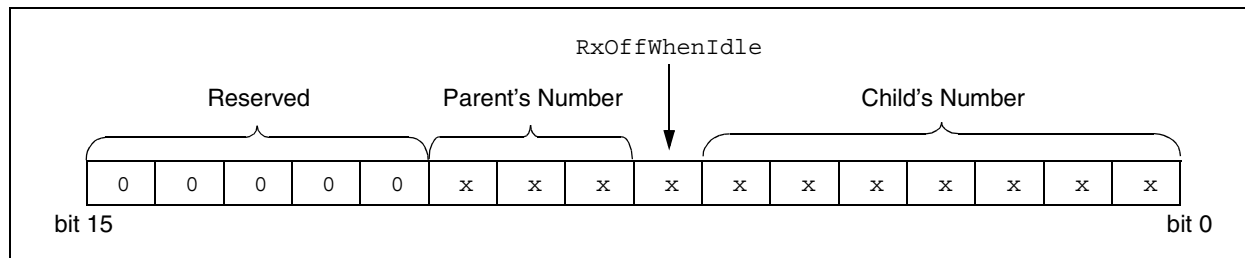
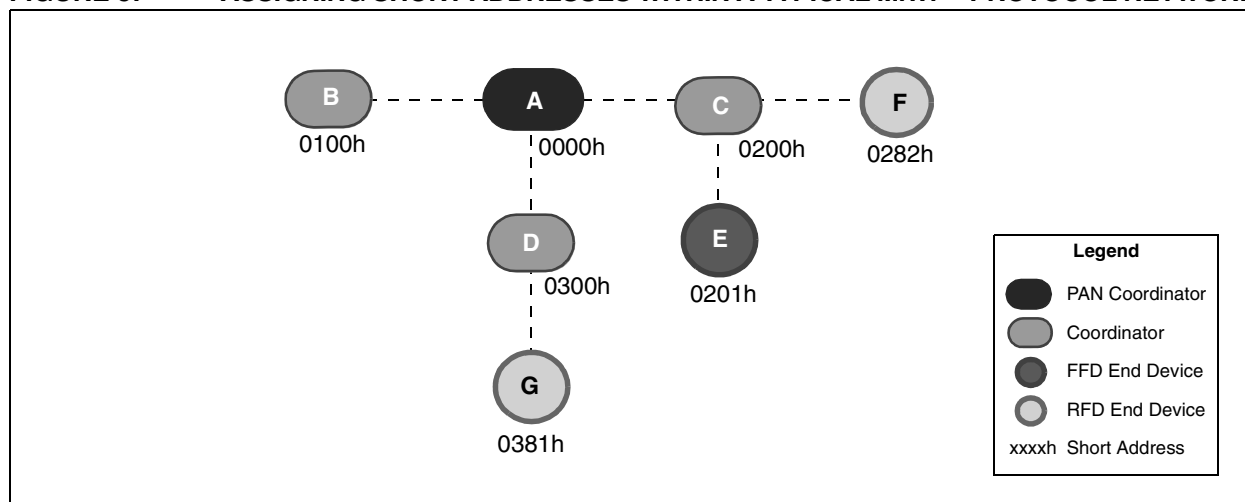


FIGURE 5: ASSIGNING SHORT ADDRESSES WITHIN A TYPICAL MIWi™ PROTOCOL NETWORK



MIWI PROTOCOL MESSAGING

Once a network has been formed, the next major concern is how to send messages through the network. Any device that is a member of a MiWi protocol network will use its short address to communicate through the network. This short address helps other devices in the network determine the location of the node and how to route to that device. Devices that have formed P2P relationships with other devices, however, use their long addresses to communicate.

Packet Format

The MiWi protocol uses the IEEE 802.15.4 MAC layer packet format for all of its packets. Nodes that have joined the network will use the Short Address mode provided in the IEEE specification. P2P nodes will use both Long Address mode for both source and destination. The packets should be constructed according to Section 7.2 of the IEEE 802.15.4 specification.

Above this layer resides the MiWi protocol header that contains information needed for routing and packet processing. This header format is shown in Figure 6. It is comprised of the following components:

- **Hops:** The number of hops that the packet is allowed to be retransmitted (00h means don't retransmit this packet – 1 byte).
- **Frame Control:** The Frame Control field is a bit-map that defines the behavior of this packet. The individual bits are defined in Table 4 (1 byte).
- **Dest PANID:** The PANID of the final destination node (2 bytes in the MiWi protocol).
- **Dest Short Address:** The final destination's short address (2 bytes).
- **Source PANID:** The PANID of the node that originally sent the packet (2 bytes).
- **Source Short Address:** The short address of the node that originally sent the packet (2 bytes).
- **Sequence Number:** A sequence number that can be used to track the status of packets as they travel through the network (1 byte).
- **Report Type:** The grouping of the message contained in this packet. All Stack generated packets have a Report Type of 00h. All user-defined reports are 01h through FFh. Stack messages and services are described later in this document (1 byte).
- **Report ID:** The type of message contained in this packet (1 byte).

Note: Refer to the section on “**MiWi Protocol Security**” for the security header format.

FIGURE 6: MiWi™ PROTOCOL PACKET HEADER FORMAT

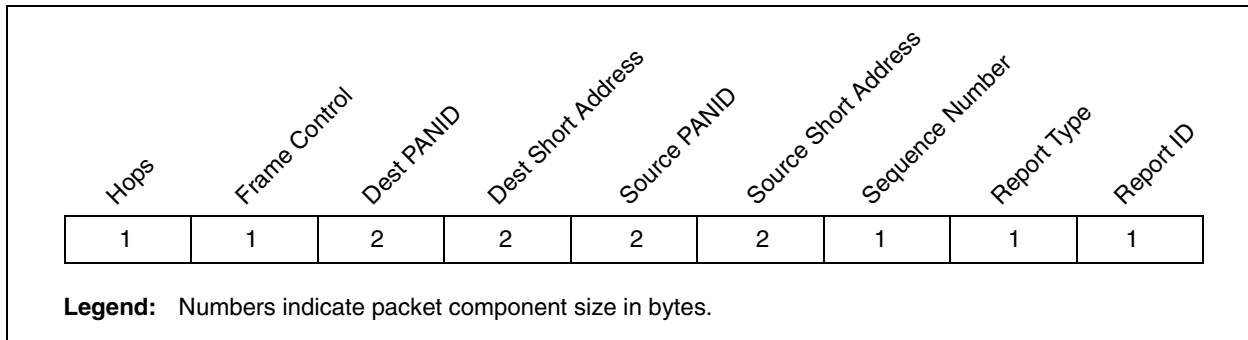


TABLE 4: FRAME CONTROL BIT FIELD

0	0	0	0	0	x	1	0
r	r	r	r	r	ACKREQ	INTRCLST	ENCRYPT
bit 7						bit 0	

- bit 7-3 **Reserved:** Maintain as '0' in this implementation
- bit 2 **ACKREQ:** Acknowledge Request bit
When set, the source device requests an upper layer Acknowledgement of receipt from the destination device.
- bit 1 **INTRCLST:** Intra Cluster bit
Reserved in this implementation, maintain as '1'.
- bit 0 **ENCRYPT:** Encrypt bit
When set, data packet is encrypted at the application level.

Note: Abbreviated bit names are for convenience of display only; they are not an official part of IEEE 802.15.4™.

Routing

Routing in wireless networks can be a very difficult and resource intensive task. The MiWi protocol solves this problem by using the address allocation to indicate the parent of the device you want to send the packet to, and by using the already provided IEEE services to help exchange and relay routing information in the network.

LEARNING ABOUT NEIGHBORING COORDINATORS

One of the tasks of a routing algorithm is determining the next hop for any outgoing packet. The MiWi protocol uses the IEEE network join mechanism, in addition to regular network traffic, to discover these paths. When any device is joining onto the network, it first sends out a beacon request packet. All of the coordinators that hear the beacon request packet send out a beacon packet informing neighboring devices of their network information.

In the MiWi protocol, three bytes of additional information are attached to the beacon payload to assist with routing:

- **Protocol ID (1 byte):** This helps distinguish MiWi protocol networks from other IEEE 802.15.4 networks that may be operating in the same radio range. Protocol ID should always be 4Dh.
- **Version Number (1 byte):** The version number of the specification. Stacks based on this specification should use 10h.
- **Local Coordinators (1 byte):** This field is a bitmap that indicates which coordinators are currently visible by the coordinator that is sending the beacon. Each bit position directly represents one of 8 possible coordinators. Bit 0 is 0000h (the PAN coordinator). Bit 1 indicates that this coordinator can talk directly to 0100h, and so on.

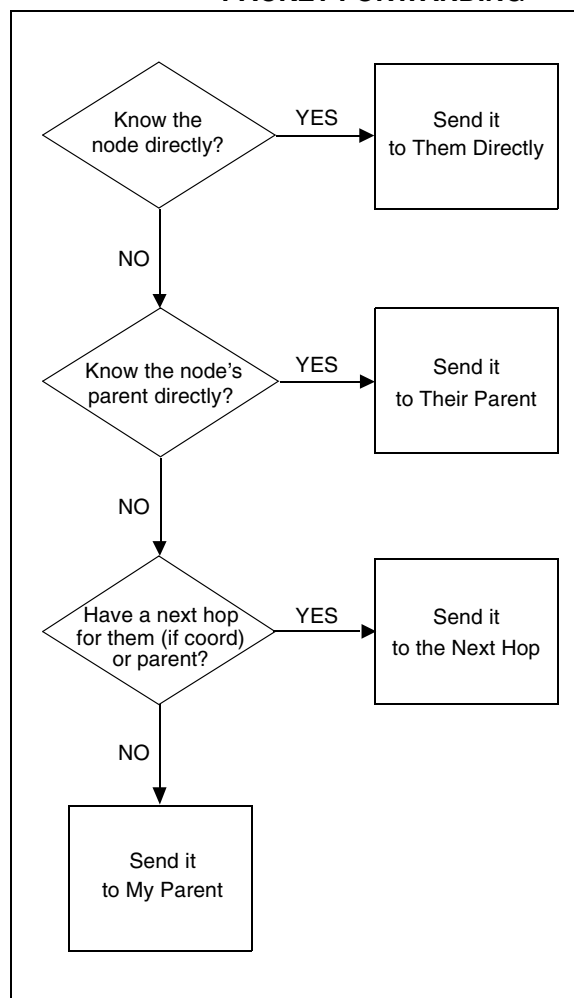
For example: Coordinator 0x200 is capable of talking to 0x500 and the PAN coordinator. The Local Coordinators field would be 'b00100101'.

Through the Local Coordinators field of the beacon payload, all of the coordinators on the network will learn about various possible routes to all of those nodes without having to send out unique requests.

ROUTING TO OTHER DEVICES

Routing in MiWi protocol networks becomes easy once we have knowledge of the neighboring coordinators, as well as what those coordinators can see. Sending a packet to another node follows the logic shown in Figure 7.

FIGURE 7: DECISION TREE FOR PACKET FORWARDING



Broadcast Messages

When a MiWi protocol network coordinator receives a broadcast packet, it will rebroadcast the packet as long as the Hops counter (the first byte of the header) is not equal to zero. Broadcast packets are not forwarded to end devices.

Broadcast packets should always have their ACK request bits (in both the MiWi protocol header and the MAC header) set to '0'. Coordinators that receive broadcast packets should then process the packet after retransmitting it.

MiWi Protocol Reports

The MiWi protocol transports packets between devices by using special packets called reports. The protocol allows for the implementation of up to 256 Report Types, and up to 256 separate Report IDs for each Report Type. The Report ID is the specification function of the packet.

Report Type 00h is reserved for MiWi Wireless Networking Protocol Stack packets, which have packet payload that is directed to the Stack. For example, a MiWi protocol ACK has a Report Type of 00h (because it is a Stack packet) and a Report ID of 30h. All other Report Types are available for the user.

The Report Type and Report ID are defined in the packet header, as previously described in the “**Packet Format**” section. The size and contents of the payload of a report depends on the particular Report ID. In this implementation of the MiWi protocol, the payload size varies from 0 bytes (i.e., sending a packet with a specific Report Type/ID is essentially the entire message) to 10 bytes, with multi-byte payloads being delivered Least Significant Byte (LSB) first. A list of the implemented reports in the current version of the protocol is provided in Table 5, with detailed descriptions immediately following.

TABLE 5: REPORTS IMPLEMENTED IN THE MIWi™ PROTOCOL

Report Type	Report ID	Name
00h	10h	OPEN_CLUSTER_SOCKET_REQUEST
	11h	OPEN_CLUSTER_SOCKET_RESPONSE
	12h	OPEN_P2P_SOCKET_REQUEST
	13h	OPEN_P2P_SOCKET_RESPONSE
	20h	EUI_ADDRESS_SEARCH_REQUEST
	21h	EUI_ADDRESS_SEARCH_RESPONSE
	30h	ACK_REPORT_TYPE
01h-FFh	00h-FFh	Available for use

OPEN_CLUSTER_SOCKET_REQUEST

Report Type (1 byte)	Report ID (1 byte)	Requesting EUI Address (8 bytes)
00h	10h	The EUI of the initiating device.

The destination address of the MiWi protocol header should be the PAN coordinator (0000h). The source address of the MiWi protocol header should be the

address of the device that is initiating the request. The Requesting EUI Address field specifies the EUI of the device initiating the request, LSB first.

OPEN_CLUSTER_SOCKET_RESPONSE

Report Type (1 byte)	Report ID (1 byte)	Resulting EUI Address (8 bytes)	Resulting Short Address (2 bytes)
00h	11h	The EUI of the resulting device.	The short address of the resulting device.

The destination address of the MiWi protocol header should be the original requesting device. The source address should be the PAN coordinator (as this packet will only originate from the PAN coordinator). The Resulting EUI Address field specifies the EUI of the device that responded to the request (LSB first). Note that this is a different address than the MiWi protocol destination address.

The Resulting Short Address field is the short address of the device that responded to the request. With the combination of EUI and short address sent to both requesting nodes, they should be able to communicate on the network and find each other if either of them happens to move in the network. Once the OPEN_CLUSTER_SOCKET_RESPONSE is sent out, the PAN coordinator will no longer maintain any of the socket information.

OPEN_P2P_SOCKET_REQUEST

Report Type (1 byte)	Report ID (1 byte)
00h	12h

Both the source and destination information in the MiWi protocol header should be FFFFh. The Hops field of the MiWi protocol header should be 00h in order to prevent rebroadcast of the packet. The MAC level source and

destination PANID should be FFFFh. The MAC destination short address should be FFFFh. The MAC level source address should be Long Address mode.

OPEN_P2P_SOCKET_RESPONSE

Report Type (1 byte)	Report ID (1 byte)
00h	13h

Both the source and destination information in the MiWi protocol header should be FFFFh. The Hops field of the MiWi protocol header should be 00h. The MAC level

source and destination PANID should be FFFFh. The MAC destination and source addresses should be both long addresses (EUIs).

EUI_ADDRESS_SEARCH_REQUEST

Report Type (1 byte)	Report ID (1 byte)	Search EUI Address (8 bytes)
00h	20h	The EUI of the device that is being searched for.

The destination short address and PANID of the MiWi protocol header should be the broadcast address, FFFFh. The source address and PANID of the MiWi protocol header should be the address information that is requesting the search. On reception of this packet, a coordinator in the network will rebroadcast this packet if

the number of hops is more than 00h. The coordinator will decrement the Hops counter before rebroadcasting the packet. The coordinator will not change the value of the MiWi protocol sequence number when broadcasting the packet.

EUI_ADDRESS_SEARCH_RESPONSE

Report Type (1 byte)	Report ID (1 byte)	Search EUI Address (8 bytes)	Search Results PANID (2 bytes)	Search Results Short Address (2 bytes)
00h	21h	The EUI of the device that is being searched for.	The resulting device's PANID.	The resulting device's short address.

The EUI_ADDRESS_SEARCH_RESPONSE should be unicast back to the address of the device that originally sent the request (the device mentioned in the MiWi protocol source fields of the request packet).

ACK_REPORT_TYPE

Report Type (1 byte)	Report ID (1 byte)
00h	30h

The MiWi protocol source address of the MiWi protocol ACK packet should equal the MiWi protocol destination address of the packet that requires Acknowledgement. The MiWi protocol destination address of the MiWi

protocol ACK packet should equal the MiWi protocol source address of the packet that requires Acknowledgement.

STACK MESSAGES AND SERVICES

Providing address allocation and routing services are just the beginning of a wireless network solution. In addition to these basic features, the MiWi protocol provides other optional services that assist developers to more rapidly reach a solution. These services include dynamically creating connections between two devices without having to know any information about the device ahead of time (i.e., sockets), and the ability to search the network for a specific device's long address.

Discovering Nodes in the Network by EUI

When two nodes communicate over a MiWi protocol network, they use their short addresses. If the network topology ever changes, it is useful to be able to find that device again. Because the short address of a device is assigned to it by its parent, the short address of the destination device may have changed. If this is the case, then the new short address must be discovered before communication can be re-established. Unlike the short address, the EUI of a device never changes and is globally unique. If one device knows another device's EUI, it will be able to distinguish that device from any other device. Searching the network for a specific EUI then becomes important in reestablishing communication with nodes that have moved. The MiWi protocol provides such a feature to search the network for a specific EUI.

There are two Stack packets that are defined in order to assist with searching for a specific EUI on the network: `EUI_ADDRESS_SEARCH_REQUEST` and `EUI_ADDRESS_SEARCH_RESPONSE`. The Search Request is unicast to the first coordinator (Figure 8, sequence 1) and then broadcast among the coordinators into the network with the EUI of the device that needs to be located (sequence 2). It is repeated by all of the coordinators until the Hops counter dies (sequence 3).

If one of the coordinators currently has this device as a child, it returns a `EUI_ADDRESS_SEARCH_RESPONSE` packet with the device's EUI and its short address. The `EUI_ADDRESS_SEARCH_RESPONSE` is sent unicast, node by node, back to the MiWi protocol source address of the packet that originally sent the packet (sequence 4).

Opening a Socket to a Device

Another feature that may be important to some networks is the ability to dynamically form communication links on the network. This is useful in networks where preconfiguration of nodes needs to be minimal.

As an example, a user may wish to add a new light switch to a lighting control system. How does that light switch know which light to send its packets to? One

way would be to have some type of interface where the device installer manually programs controller and target addresses into the devices.

CLUSTER SOCKET

A more dynamic method would be to have a push button on both the light and the light switch. You first press the button on the light, and then the light switch, within a specified time interval. This lets these two devices know that they need to communicate with each other. This allows for dynamic run-time changes in the behavior of the network in a simple, user-friendly method.

The MiWi protocol defines this dynamically formed communication link as a cluster socket. A cluster socket exists between two nodes that are members of the network and is formed based on the short address.

In the previous example, pushing the button on the light switch sends an `OPEN_CLUSTER_SOCKET_REQUEST` to the PAN coordinator with that device's information (Figure 9, sequence 1). This notifies the PAN coordinator that the device is looking for someone to talk to. The PAN coordinator keeps that request open for an amount of time that is specified by the application. If a similar request comes from the light (sequence 2), the PAN coordinator combines the short address information from both devices into one `OPEN_CLUSTER_SOCKET_RESPONSE` and sends it back to the switch and the light (sequence 3). Finally, the PAN coordinator removes the open socket request. If the PAN coordinator doesn't hear a second open socket request within the specified amount of time, then it will terminate the open socket request without sending a response to the requesting node.

When the devices at F and G receive the `OPEN_CLUSTER_SOCKET_RESPONSE` report, they can examine the payload of that packet and determine if that device is, in fact, the device that they wish to talk to. This is an application layer decision; it is not provided by the Stack.

P2P SOCKET

Though P2P devices only talk to one device and they don't talk through the network, they still may require a dynamic connection. P2P devices must have a different algorithm from the cluster sockets since they are not members of a network.

When a P2P device wants to talk to a P2P socket with a different node, it first broadcasts an `OPEN_P2P_SOCKET_REQUEST` frame which contains the full device's EUI (Figure 10, sequence 1). Any device that hears that request that wishes to talk to that device sends an `OPEN_P2P_SOCKET_RESPONSE` frame that contains its own EUI address (sequence 2). This informs the first device that it wishes to talk to that device. In the current implementation, only coordinators are capable of receiving the `OPEN_P2P_SOCKET_REQUEST`.

FIGURE 8: SEQUENCE FOR EUI ADDRESS SEARCH REQUEST AND RESPONSE

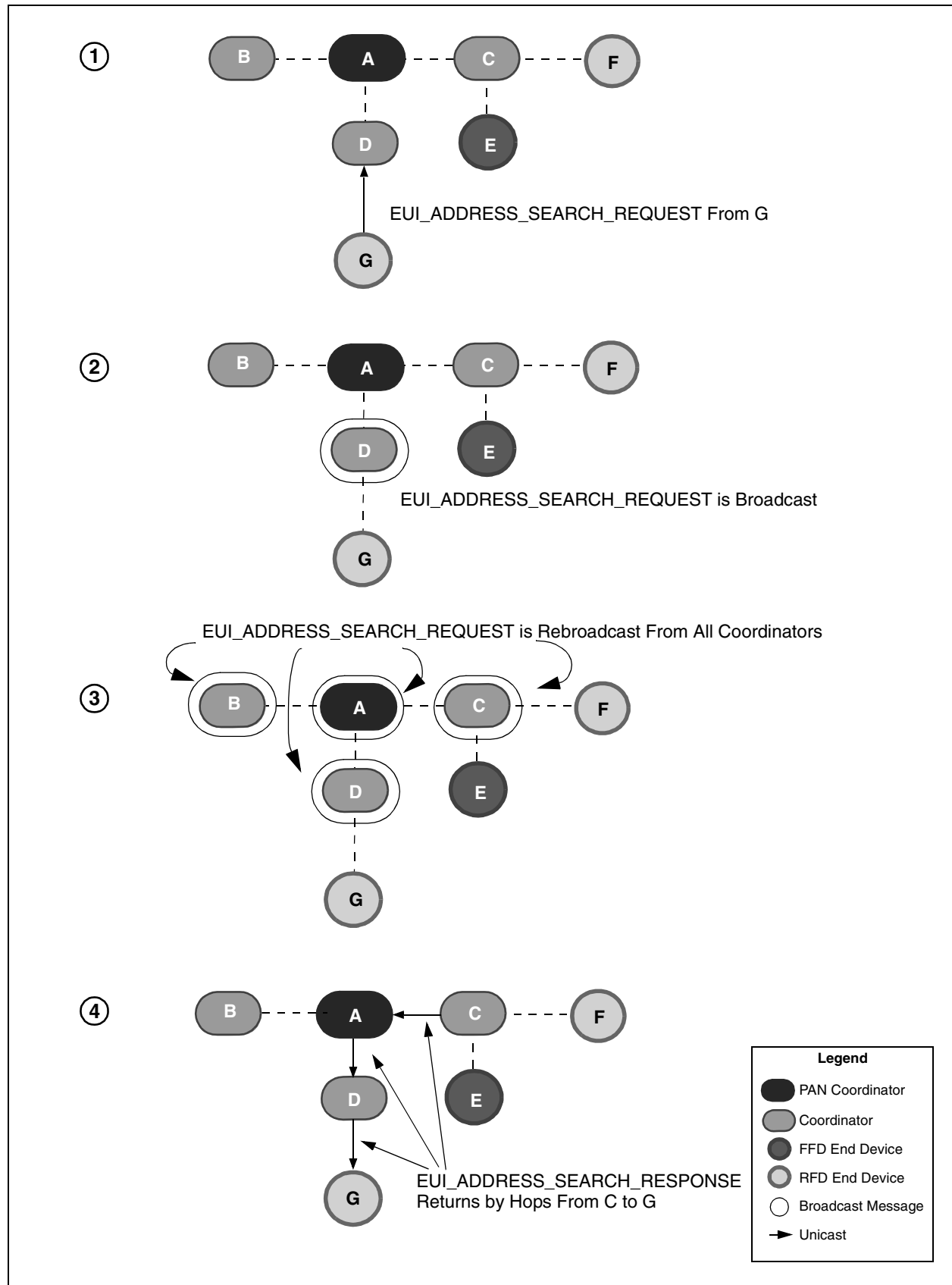


FIGURE 9: SEQUENCE FOR OPEN SOCKET REQUEST AND RESPONSE

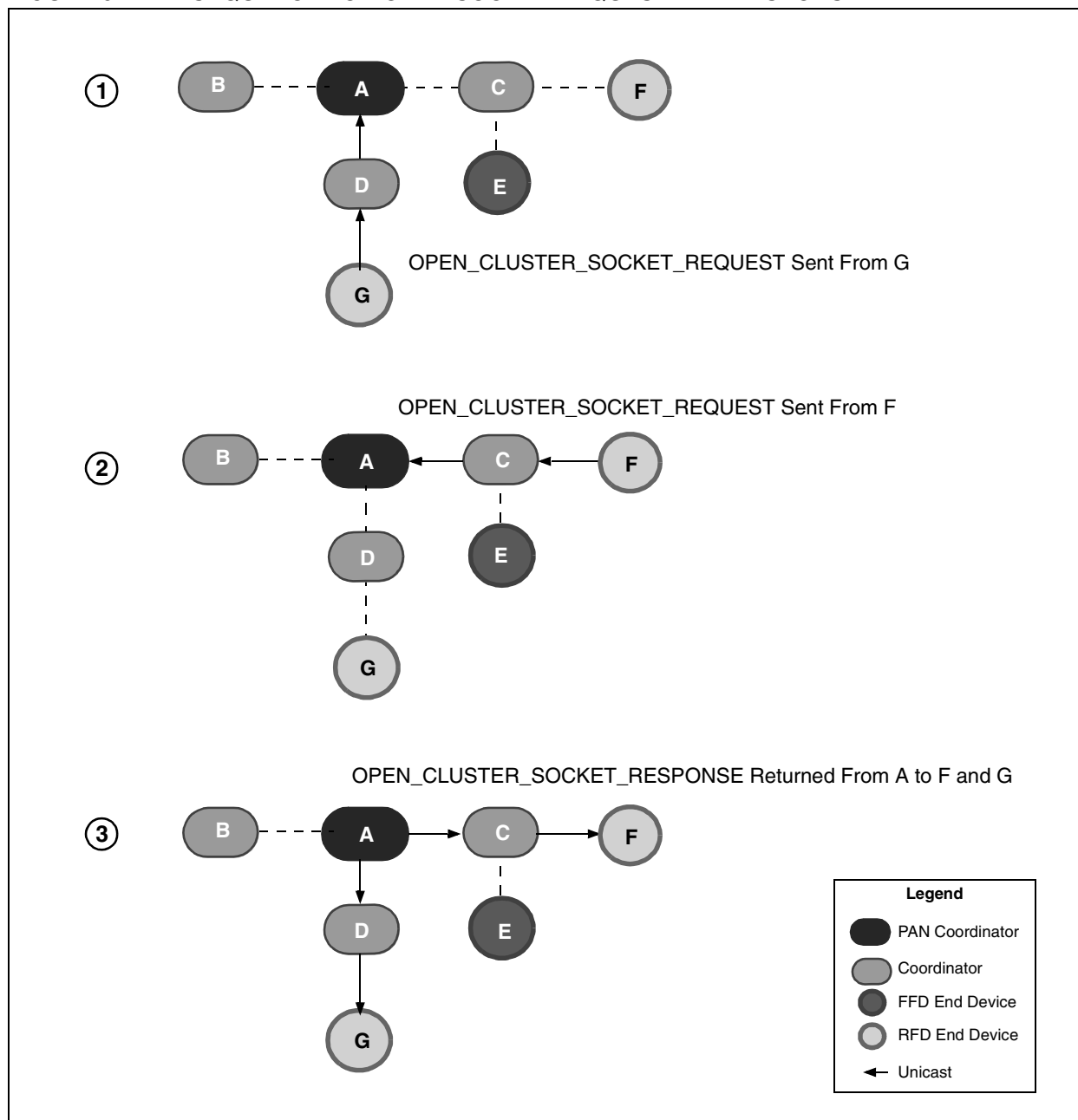
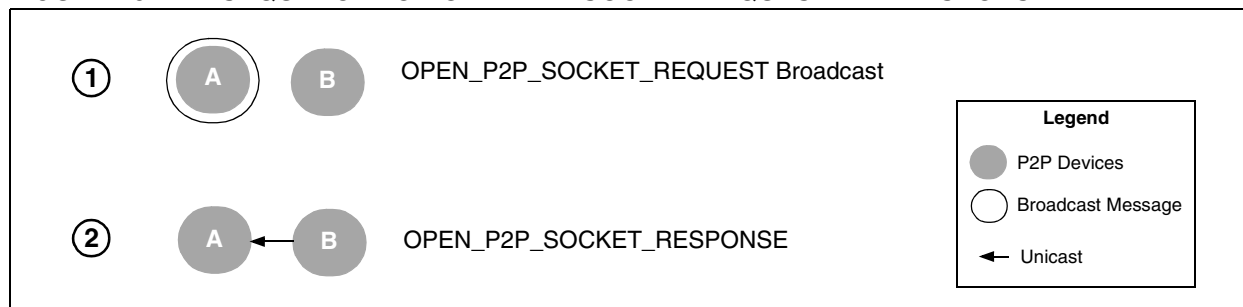


FIGURE 10: SEQUENCE FOR OPEN P2P SOCKET REQUEST AND RESPONSE



MiWi PROTOCOL SECURITY

The MiWi protocol follows the MAC security definition specified in IEEE 802.15.4. All seven of the security modes defined in the specification are supported in the MiWi protocol. These can be categorized into three groups:

- AES-CTR mode encrypts MiWi protocol payload. Attacker will not understand the content of the MiWi protocol packet without a key. This security mode cannot verify frame integrity or the content of the MiWi protocol header, and most importantly, the original source address of the packet.

- AES-CBC-MAC modes ensure the integrity of the MiWi protocol packet. The Message Integrity Code (MIC) attached to the packet ensures that the packet, including the header and payload, have not been modified in any way during transmission. The size of the MIC is determined by the particular mode; larger MICs provide stronger protection. The MiWi protocol packet payload is not encrypted in these modes.
- AES-CCM modes combine the previous two security modes to ensure both the integrity of the frame and encrypt the MiWi protocol payload.

Security mode, 00h, which specifies no security, is essentially the MiWi Wireless Networking Protocol Stack with the security mode turned off. The capability of each of the security modes can be found in Table 6.

TABLE 6: IEEE 802.15.4™ SECURITY MODES

Security Mode		Security Services				MIC (Bytes)
Identifier	Name	Access Control	Data Encryption	Frame Integrity	Sequential Freshness	
01h	AES-CTR	X	X		X	0
02h	AES-CCM-128	X	X	X	X	16
03h	AES-CCM-64	X	X	X	X	8
04h	AES-CCM-32	X	X	X	X	4
05h	AES-CBC-MAC-128	X		X		16
06h	AES-CBC-MAC-64	X		X		8
07h	AES-CBC-MAC-32	X		X		4

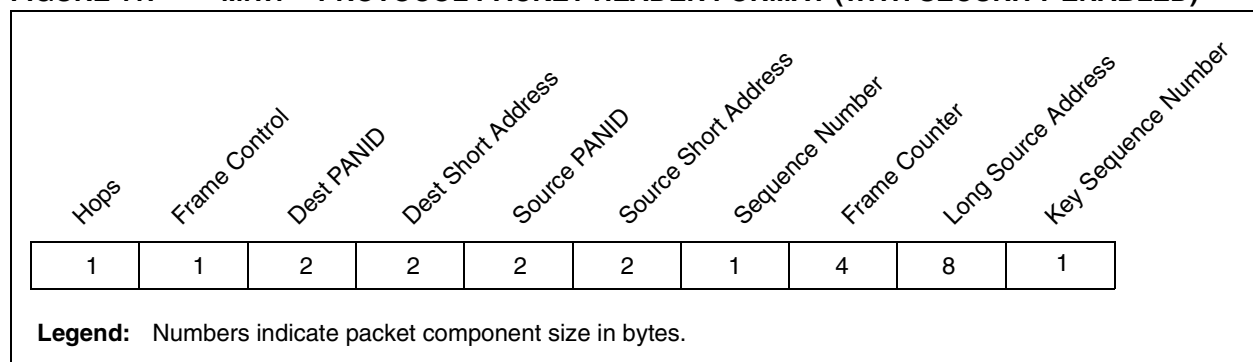
The Stack provides security support in the MiWi protocol layer. When security is turned on, the security bit (bit 0) of the Frame Control byte is set to '1'. Once security is turned on, the Stack secures all packets from the MiWi protocol layer. The MiWi protocol header (except Hops, Report Type and Report ID) is used as authentication data for all security modes, except mode 01h. The MiWi protocol Report Type and Report ID are secured, as well as the report payload. Additional details on the IEEE 802.15.4 security suite can be found in Section 7.6 of the standard.

When security is enabled, the Stack adds three new items to the packet header immediately before the data payload:

- the frame counter
- the source long address
- the key sequence number

The format of the MiWi protocol packet header, with security enabled on, is shown in Figure 11.

FIGURE 11: MiWi™ PROTOCOL PACKET HEADER FORMAT (WITH SECURITY ENABLED)



When security is used, between 13 and 29 additional bytes are required for the auxiliary security header and the MIC, depending on the combination of security mode and secured layer. Users will need to balance the security needs, and the impact on the data payload size (and associated performance impact), associated with the combination of security settings.

The MiWi protocol security checks the frame counter to avoid repeat attacks. Only MiWi protocol packets from a node's family member (i.e., parent and children) are checked for frame counter freshness. The reason is that only family members are able to know if each other join or leave the network. Any packets from family members are required to have the frame counter not smaller than the frame counter stored; otherwise, the packet is dropped. Any parent will reset the corresponding incoming frame counter stored once the corresponding child joins or rejoins the network.

The MiWi protocol security assumes that all the security information, including the 16-byte security key, key sequence number and security mode, are preconfigured in the Stack. The recommended way to do this is to use the ZENA analyzer tool to configure the security information and create the `MiWiDefs.h` file used in the application. This way, the security information is stored in program memory and cannot be modified during the run. This is the safest way to implement the security features, since no security key will be transferred over the air.

SETTING UP A MIWI PROTOCOL STACK PROJECT

Using the MiWi Wireless Networking Protocol Stack requires the user to first install the MiWi protocol source and definition files onto the development system (PC). The installer file automatically creates the required directory structure for MiWi protocol projects. The required files are organized into three separate folders, listed in Table 7. Users will need to configure their development environments to include these directories and all listed files in their projects.

USING THE ZENA™ ANALYZER AS A CONFIGURATION TOOL

To assist in the development of MiWi protocol applications, Microchip provides a low-cost network analyzer, called ZENA. The ZENA software also contains a tool to create application-specific configuration files for MiWi protocol applications. The ZENA demo software is provided free as part of the Microchip Stack for ZigBee installation (AN965) and is located in the `MpZBee` directory. Refer to AN965, “ZENA™ Wireless Network Analyzer User's Guide” (DS51606) for more information on using this tool.

The demo version of the ZENA software provides the capability of creating application-specific source files to support the Microchip Stack and analyzing previously captured wireless network traffic. The full-featured version of ZENA software, which includes the ability to

capture real-time wireless network activity, is available as a separate kit and includes an RF sniffer that can be connected to a PC through a USB port.

Note: The ZENA demo software is used to create the file, `MiWiDefs.h`, which is critical to the configuration of the Stack.

TABLE 7: MiWi™ PROTOCOL STACK SOURCE FILES AND THEIR LOCATIONS

Subdirectory	File Name	Function
/common	Compiler.h	Compiler-specific definitions.
	Console.c	
	Console.h	
	GenericTypeDefs.h	Generic constants and type definitions.
	MSPI.c	SPI interface code.
	MSPI.h	
	SymbolTime.c	Performs timing functions for the MiWi Wireless Networking Protocol Stack.
	SymbolTime.h	
/demo	Demo.mcp	MPLAB® IDE project file.
	Main.c	The main demo code
	MiWiDefs.h	Generated by ZENA™ software. Contains application-specific information.
/MiWi Stack	MiWi.c	The MiWi Wireless Networking Protocol Stack.
	MiWi.h	
	MRF24J40.h	Interface definitions for the MRF24J40.

USING THE STACK

The first step in getting started with a MiWi protocol application is to download the source code that corresponds to this application note and the AN965 documentation and source files. Inside the AN965 source distribution is the demo version of the ZENA software. The demo version of the ZENA software provides the tools required to generate the `MiWiDefs.h` source file that is needed by the Stack.

The MiWi Wireless Networking Protocol Stack is written mainly using cooperative multitasking. This means that while the Stack is performing long tasks, such as

searching or joining networks, it returns the control of the CPU to the user application. It is the responsibility of the user application to return control back to the Stack whenever CPU time is available.

Depending on the duration of the required tasks, the user application may need to be split into smaller tasks with calls into the Stack between the executions of these blocks. The code in Example 1 shows one method of doing this.

EXAMPLE 1: SWITCHING BETWEEN THE MiWi™ PROTOCOL STACK AND APPLICATION BLOCKS

```
#define MAX_APPLICATION_STATE 1
BYTE applicationState=0;
While(1)
{
    MiWiTasks();
    Switch(applicationState)
    {
        case 0:
            //first application block here
            break;
        case 1:
            //second block of application code here
            break;
    }
    applicationState++;
    if(applicationState>MAX_APPLICATION_STATE)
    {
        applicationState=0;
    }
}
```


DATA STRUCTURES

Apart from the MiWi protocol API itself, there are two data structures that are shared between the Stack and the user application: the network table and the TX/RX buffers.

Network Table

The network table plays a critical role in the operation of the MiWi Wireless Networking Protocol Stack. The Stack uses the network table to store information about available networks, its parent, its children, as well as other nodes that it is communicating with. The results of many of the socket requests result in an entry in the network table. The size of the network table is config-

urable through the ZENA analyzer configuration software, or through manual modification of the `MiWiDefs.h` file.

The network table is used to store two types of information: information about networks and information about neighbors. Before looking at a network table entry, the `networkTable[].status.isValid` bit should be checked to see if that particular entry in the network table is valid.

The `networkTable[].status.NeighborOrNetwork` bit specifies if the entry is a neighboring node (`= 1`) or a network (`= 0`). If the entry is a valid neighbor, then it will have the attributes listed in Table 8. If the entry is a valid network entry, then it will have the attributes listed in Table 9.

TABLE 8: NETWORK TABLE ATTRIBUTES FOR VALID NEIGHBORS

Network Status	Description
<code>networkTable[].status.directConnection</code>	Can communicate directly with this node without having to route through another device.
<code>networkTable[].status.longAddressValid</code>	The <code>networkTable[].info.LongAddress[]</code> field is valid.
<code>networkTable[].status.shortAddressValid</code>	The <code>networkTable[].ShortAddress</code> field is valid.
<code>networkTable[].status.P2PConnection</code>	This device is in a P2P relationship with me.
<code>networkTable[].status.RXOnWhenIdle</code>	This device has its transceiver on when it is idle if this bit is '1', off if this bit is '0'.
<code>networkTable[].status.isFamily</code>	This device is related to this node (either a child or the parent).
<code>networkTable[].PANID</code>	The PANID of the device.
<code>networkTable[].ShortAddress</code>	The short address of the device (valid only if <code>networkTable[].status.shortAddressValid</code> is '1').
<code>networkTable[].info.LongAddress[]</code>	The long address of the device (valid only if <code>networkTable[].status.longAddressValid</code> is '1').

TABLE 9: NETWORK TABLE ATTRIBUTES FOR VALID NETWORKS

Network Entry	Description
<code>networkTable[].PANID</code>	The PANID of the device.
<code>networkTable[].ShortAddress</code>	The short address of the device that you learned about this network from.
<code>networkTable[].info.networkInfo.Channel</code>	Channel that this network is located on.
<code>networkTable[].info.networkInfo.sampleRSSI</code>	The RSSI value of the beacon frame received that described this network.
<code>networkTable[].info.networkInfo.Protocol</code>	The protocol version of this network (should equal 4Dh for this specification).
<code>networkTable[].info.networkInfo.Version</code>	The implementation version of the protocol.
<code>networkTable[].info.networkInfo.flags.PANcoordinator</code>	This bit is set if the device is the PAN coordinator.
<code>networkTable[].info.networkInfo.flags.associatinPermit</code>	This bit is set if the device is allowing devices to join to it. This bit is clear if the device is not allowing nodes to join.

Receive and Transmit Buffers

IEEE 802.15.4 defines the packet size as 127 bytes. The MRF24J40 has 128-byte transmit and receive buffers that allow for full packet reception and transmission. The code implements RAM buffers to assist the MRF24J40 transceiver: one for transmission (TX buffer) and one for reception (RX buffer).

Many networks only need to transmit and receive small amounts of data. In these networks, the burden of large RAM buffers can be reduced by artificially limiting the maximum packet size. This is accomplished by reducing the size of the transmit and receive buffers. The ZENA analyzer provides a graphical interface to select the size of these buffers or they can be manually adjusted in the `MiWiDefs.h` file.

There are two functions that are required to write data to the transmit FIFO: `WriteData()` and `FlushTx()` (to clear the FIFO). The first byte written to the FIFO should be the Report Type. The next byte of the FIFO should be the Report ID. All of the application data can be populated after that point.

When the `RxPacket()` function returns TRUE, this indicates that a packet arrived that was not directed towards the Stack. The `RxSize` variable will contain the number of bytes available and `pRxData` is a pointer to the data.

EXAMPLE 2: TRANSMITTING A PACKET

```
WriteData(0x55);           // Report Type
WriteData(0xAA);           // Report ID
WriteData(0xFF);           // Data
SendReportByHandle(0x00,FALSE); // send the packet to device that is
                               // the first entry in the network table
```

EXAMPLE 3: RECEIVING A PACKET

```
if(RxPacket())
{
    switch(*pRxData++)           //report type
    {
        case USER_REPORT_TYPE:
            switch(*pRxData++)   //report id
            {
                case LIGHT_REPORT:
                    switch(*pRxData++) //first byte of payload
                    {
                        case LIGHT_ON:
                            LATAbits.LATA1 = 1;
                            break;
                    }
                break;
            default:
                while(RxSize--)
                {
                    PrintChar(*pRxData++); //unknown data, print it out
                }
                break;
            }
    }
}
```

MiWi PROTOCOL STACK OPERATIONS

The basic operations of the MiWi Wireless Networking Protocol Stack are executed through a simple set of function calls which are described starting on page 20. The operations themselves are briefly described below.

Searching for Networks

Searching for available networks to join to is the first networking function every device must perform. The `DiscoverNetworks()` function will cause the device to search all of the channels specified in the available channels array for a network and populate the network table with this information. The function, `SearchingForNetworks()`, returns TRUE while the device is still searching for a network.

Joining a Network

Once the search for networks is complete a device will have to traverse the resulting list of networks to determine if any of the networks are acceptable to join. If an adequate network is discovered, a device can attempt to join that network. The function, `JoinNetwork()`, takes in the handle of the network that the device wishes to join. The `AttemptingToJoinNetwork()` function will return TRUE as long as the Stack is still trying to join to the network. Once the `AttemptingToJoinNetwork()` function returns FALSE, the `MemberOfNetwork()` function indicates if the device successfully joined the network.

Forming a Network

A device that is capable of becoming a PAN coordinator may determine that there are no suitable networks available and form its own network. This is done using the `FormNetwork()` function. `FormNetwork()` forms a network with the specified PANID. If the input PANID is FFFFh, `FormNetwork()` randomly selects a PANID.

Sending and Receiving Messages

There are several different methods for sending packets in a MiWi protocol network. For nodes that have an entry in the network table, the most simple way to send the packet is to call the `SendReportByHandle()` function.

If the short address of the device is known then a packet can be sent to that device using the `SendReportByShortAddress()` function.

If the device's long address is known, then it may be possible to send the packet to that device with the `SendReportByLongAddress()` function. This function will only send the packet to devices that it knows it can directly communicate with. If the device is not in the network table, then the `SendReportByLongAddress()` function will fail and a `DiscoverNodeByEUI()` needs to be sent in order to find the location of that device.

For information on receiving a message, see “**Receive and Transmit Buffers**”.

Requesting Data with an RFD

RFD end devices are slightly different from other devices in the network in that they are capable of turning off their transceivers while they are Idle. This requires their parents to buffer messages for the RFD end device until it requests for the data through the data request protocol of IEEE 802.15.4. There are two functions required to handle the data request process in the Stack: `CheckForData()` and `CheckForDataComplete()`. `CheckForData()` will cause the data request process to start. `CheckForDataComplete()` indicates when that process completes (in the even that the coordinator had no data for the device).

Discovering a Node on the Network by its EUI

It is frequently the case that a device knows the long address (EUI) of the device it wishes to talk to but does not know where on the network that device is located. The MiWi protocol provides functionality for finding these devices as described in the “**Stack Messages and Services**” section. `DiscoverNodeByEUI()` will assist in finding a node on the network by its EUI. The function expects that the long address of the target device has been loaded into the `tempLongAddress` variable.

Creating a Socket

In many applications, devices that need to talk to one another may not know the device that they wish to talk but rather wish to form these links dynamically at run time. This functionality is described in the “**Stack Messages and Services**” section.

Several functions are required for opening a socket. The first function required is `OpenSocket()` which actually attempts to open the socket. After the request to open the socket is issued then the results of that request need to be checked. Other helper functions that are required to determine if the socket request is complete, if it was successful, and what the results were if it was successful (`OpenSocketComplete()`, `OpenSocketSuccessful()` and `OpenSocketHandle()`).

MIWI PROTOCOL STACK FUNCTIONS

AddNodeToNetwork

This function adds a node to a network table. The `tempLongAddress`, `tempShortAddress`, `tempPANID`, and `tempNodeStatus` variables need to be set to before calling this function. `tempNodeStatus` mirrors the `networkTable.status` format. This should be set to the desired value.

Syntax

```
BYTE AddNodeToNetworkTable( void );
```

Inputs

None

Outputs

BYTE: The index of the network table entry where the device was inserted. FFh indicates that the requested device couldn't be inserted into the table.

Notes

None

AttemptingToJoinNetwork

This function indicates if the device is still in the process of joining a network.

Syntax

```
BOOL AttemptingToJoinNetwork( void );
```

Inputs

None

Outputs

TRUE: The Stack is still in the process of joining a network.

FALSE: The last association request is complete.

Notes

None

AttemptingToRejoinNetwork

This function indicates if the device is still in the process of rejoining a network.

Syntax

```
BOOL AttemptingToRejoinNetwork( void );
```

Inputs

None

Outputs

TRUE: The Stack is still in the process of rejoining a network.

FALSE: The last orphan notification is complete.

Notes

None

CheckForData

This function causes an end device to poll its parent for data.

Syntax

```
void CheckForData( void );
```

Inputs

None

Outputs

None

Notes

This function is required in an end device in order to retrieve buffered data from its parent. The frequency at which this function is called is up to the application.

Example

```
if (CheckForDataComplete())
{
    SWDTEN = 1;      //enable the WDT to wake me up
    Sleep();         //goto sleep
    SWDTEN = 0;      //disable the WDT
    CheckForData();  //when I wake up do a data request
}
```

CheckForDataComplete

This function indicates that the data request completed correctly.

Syntax

```
BOOL CheckForDataComplete( void );
```

Inputs

None

Outputs

TRUE: The previous data request completed.

FALSE: There is still a data request in the works. A device should not issue another data request or go to Sleep during this time.

Notes

This function should be checked before going to Sleep or issuing another data request.

AN1066

ClearToSend

Checks to see if the transmit FIFO is ready to send data.

Syntax

```
BOOL ClearToSend( void );
```

Inputs

None

Outputs

TRUE: The transmit buffer is ready for more data. Use WriteData() to send more data.

FALSE: The transmit buffer is not ready for more data. Wait for the bit to clear itself.

Notes

None

ClearNetworkTable

Clears the specified network table entries.

Syntax

```
void ClearNetworkTable( BYTE options );
```

Inputs

options: The options selected for clearing the table. Multiple options can be selected at any point of time by ORing these options together. The available options are as follows:

Option	Description
CLEAR_ALL_ENTRIES	Clears all entries in the network table.
CLEAR_NON_DIRECT_CONNECTIONS	Clears any device that the current node doesn't directly communicate with.
CLEAR_NO_LONG_ADDRESS	Clears any neighbor entry that doesn't have a recorded long address.
CLEAR_NO_SHORT_ADDRESS	Clears any neighbor entry that doesn't have a recorded short address.
CLEAR_P2P	Clears all P2P entries out of the table.
CLEAR_NETWORKS	Clears all of the network entries out of the table.
CLEAR_NEIGHBORS	Clears all of the neighbor entries out the table.

Outputs

None

Notes

None

DiscardPacket

This function resets the Stack variables that are associated with the reception of a user packet. This prepares the Stack to receive the next available packet.

Syntax

```
void DiscardPacket( void );
```

Inputs

None

Outputs

None

Notes

None

DiscoverNetworks

Searches the channels list for available networks.

Syntax

```
void DiscoverNetworks( void );
```

Inputs

None

Outputs

None

Notes

When this function completes, the `SearchingForNetworks()` function will return FALSE.

Example

```
if(SearchingForNetworks() == FALSE)
{
    if(networkSearchStarted == FALSE)
    {
        DiscoverNetworks();
        networkSearchStarted = TRUE;
    }
}
else
{
    //the search is complete (Started but not currently searching)
    ...
}
```

DiscoverNodeByEUI

This function initiates a network discovery based on the device's EUI.

Syntax

```
void DiscoverNodeByEUI( void );
```

Inputs

None

Outputs

None

Notes

This function requires that the long address of the device that you wish to search for be preloaded into tempLongAddress. The results of the search are automatically entered into the network table of the requesting device.

Example

```
tempLongAddress[0] = 0x91;  
tempLongAddress[1] = 0x78;  
tempLongAddress[2] = 0x56;  
tempLongAddress[3] = 0x34;  
tempLongAddress[4] = 0x12;  
tempLongAddress[5] = 0xA3;  
tempLongAddress[6] = 0x04;  
tempLongAddress[7] = 0x00;
```

```
DiscoverNodeByEUI();
```

FlushTx

This function clears the data in the transmit buffer.

Syntax

```
void FlushTx( void );
```

Inputs

None

Outputs

None

Notes

This function can be used to clear any values written with WriteData() before they are sent.

FormNetwork

This function forms a network on the current channel.

Syntax

```
void FormNetwork( WORD PANID );
```

Inputs

PANID: If the input into the function is FFFFh, then the function will randomly select a PANID for the network. If the PANID is any value other than FFFFh, then the device will attempt to create a network with that PANID.

Outputs

None

Notes

The function may refuse to create the network if it determines that it already knows of a network by that PANID.

Examples

```
FormNetwork(0xFFFF);           //create a network with a random PANID

FormNetwork(0x1234);           //create a network with the PANID of 0x1234
```

JoinNetwork

This function attempts to join a specified network.

Syntax

```
void JoinNetwork( BYTE handle );
```

Inputs

handle: The handle of the network entry in the network table that the device is trying to join.

Outputs

None

Notes

The AttemptingToJoinNetwork() function will return TRUE as long as this process is still going. MemberOfNetwork() will become TRUE if it was successful.

Example

```
JoinNetwork(TheHandleOfTheBestNetwork);
```

AN1066

MemberOfNetwork

This function indicates if the device is currently a member of a network.

Syntax

```
BOOL MemberOfNetwork( void );
```

Inputs

None

Outputs

TRUE: This node is a member of a network already.

FALSE: This node has not joined onto a network yet.

Notes

None

MiWiInit

This function initializes the Stack.

Syntax

```
void MiWiInit( void );
```

Inputs

None

Outputs

None

Notes

None

MiWiTasks

This function handles all of the Stack functions, including Stack services, sending and receiving packets, etc.

Syntax

```
void MiWiTasks( void );
```

Inputs

None

Outputs

None

Notes

This function needs to be called as often as possible. The more frequent this function is called, the faster packets are processed and the more throughput the Stack can handle. This function must be called in order to receive packets and complete Stack requests.

OpenSocket

This function attempts to open a socket to other devices. This function should only be called if there is not a socket request open.

Syntax

```
void OpenSocket( BYTE mode );
```

Inputs

P2P_SOCKET: This will request the Stack to open a P2P socket.

CLUSTER_SOCKET: This request will cause the Stack to initiate a cluster socket request.

Outputs

None

Notes

The results of this socket request are found using the `OpenSocketComplete()`, `OpenSocketHandle()` and `OpenSocketSuccessful()` functions.

Examples

```
OpenSocket (CLUSTER_SOCKET) ;
```

```
OpenSocket (P2P_SOCKET) ;
```

OpenSocketComplete

This function resets the Stack variables that are associated with the reception of a user packet. This prepares the Stack to receive the next available packet

Syntax

```
BOOL OpenSocketComplete( void );
```

Inputs

None

Outputs

TRUE: The previous `OpenSocket()` request has finished.

FALSE: The previous `OpenSocket()` request is still active, either waiting for a response or a time-out.

Notes

None

Example

```
if (OpenSocketComplete())
{
    if (OpenSocketSuccessful())
    {
        myFriend = OpenSocketHandle();
    }
}
```

OpenSocketHandle

This function returns the handle into the network table of the device that was found during the last `OpenSocket` request. `OpenSocketComplete` and `OpenSocketSuccessful` should return `TRUE` before using this value. This value is no longer valid after a new open socket request is issued.

Syntax

```
BYTE OpenSocketHandle( void );
```

Inputs

None

Outputs

BYTE: The handle of the device that was discovered in the socket request.

Notes

None

OpenSocketSuccessful

This function indicates if the previous `OpenSocket` request was successful or not successful. If successful, the data is accessed via the `OpenSocketHandle()` macro.

Syntax

```
BOOL OpenSocketSuccessful( void );
```

Inputs

None

Outputs

TRUE: The previous `OpenSocket()` request was successful.

FALSE: The previous `OpenSocket()` request was unsuccessful.

Notes

None

RejoinNetwork

This function attempts an orphan notification on the current channel.

Syntax

```
void RejoinNetwork( void );
```

Description

Inputs

None

Outputs

None

Notes

`AttemptingToRejoinNetwork()` returns `TRUE` as long as this process is ongoing. `MemberOfNetwork()` will become `TRUE` if it was successful.

RxPacket

Indicates if there is a received packet for the user.

Syntax

```
BOOL RxPacket( void );
```

Inputs

None

Outputs

TRUE: There is a packet pending for the user.

FALSE: There is no RX data.

Notes

None

SearchingForNetworks

This function indicates if the device is still in the process of searching for a network to join.

Syntax

```
BOOL SearchingForNetworks( void );
```

Inputs

None

Outputs

TRUE: The Stack is still looking for a network to join. It is not a member yet, nor has it completed searching the available channels.

FALSE: The last network scan is complete.

Notes

The device will only scan the channels specified in the file `MiWiDefs.h` (generated manually or by ZENA software).

Example

```
BOOL networkSearchStarted = FALSE;
```

```
While(1)
{
    MiWiTasks();

    if(MemberOfNetwork() == FALSE)
    {
        If(SearchingForNetworks() == FALSE)
        {
            If(networkSearchStarted == FALSE)
            {
                DiscoverNetworks();
                networkSearchStarted = TRUE;
            }
        }
        else
        {
            //the search is complete (Started but not currently searching)
        }
        //pick a network here
    }
}
```

SendReportByHandle

This function sends the packet loaded in the transmit buffer to the specified device.

Syntax

```
BYTE SendReportByHandle (BYTE handle, BOOL forwardPacket);
```

Inputs

BYTE handle: The index into the network table of the device that you wish to transmit to.

BOOL forwardPacket: This input should be always set to FALSE. The Stack will call this function with a value of TRUE in order to forward packets on behalf of other nodes.

Outputs

BYTE: The MiWi protocol handle for the packet that was sent. This is a one-byte sequence number that was sent with the packet in the MiWi protocol header and will be in the corresponding ACK.

Notes

None

Example

```
WriteData (REPORT_TYPE);  
WriteData (REPORT_ID);  
WriteData (USER_DATA);  
SendReportByHandle (targetDevicesHandle, FALSE);
```

SendReportByLongAddress

This function sends the packet loaded in the transmit buffer to the specified device.

Syntax

```
BYTE SendReportByLongAddress (BYTE *pLongAddress);
```

Inputs

BYTE *pLongAddress: Pointer to the long address of the destination device

Outputs

BYTE: The MiWi protocol handle for the packet that was sent. This is a one-byte sequence number that was sent with the packet in the MiWi protocol header and will be in the corresponding ACK.

Notes

None

Example

```
{  
    BYTE TargetLongAddress[8];  
  
    TargetLongAddress[0] = 0x00;  
    TargetLongAddress[1] = 0x01;  
    TargetLongAddress[2] = 0x02;  
    TargetLongAddress[3] = 0x03;  
    TargetLongAddress[4] = 0x04;  
    TargetLongAddress[5] = 0x05;  
    TargetLongAddress[6] = 0x06;  
    TargetLongAddress[7] = 0x07;  
    SendReportByLongAddress (TargetLongAddress);  
}
```

SendReportByShortAddress

This function sends the packet loaded in the transmit buffer to the specified device.

Syntax

```
BYTE SendReportByShortAddress(WORD_VAL PANID, WORD_VAL ShortAddress, BOOL forwardPacket);
```

Inputs

WORD_VAL PANID: PANID of the device that you wish to transmit the packet to.

WORD_VAL ShortAddress: Short address of the device that you wish to send the packet to.

BOOL forwardPacket: This input should always be set to FALSE.

Outputs

BYTE: The MiWi protocol handle for the packet that was sent. This is a one-byte sequence number that was sent with the packet in the MiWi protocol header and will be in the corresponding ACK.

Notes

Note that this version of the specification doesn't support inter-PAN routing, so the PANID of the destination device must equal the PANID of the device that is transmitting the packet.

Example

```
WriteData(REPORT_TYPE);  
WriteData(REPORT_ID);  
WriteData(USER_DATA);  
SendReportByShortAddress(targetPANID, targetShortAddress, FALSE);
```

TickGet

This function returns the current time as a function of system ticks.

Syntax

```
TICK TickGet( void );
```

Inputs

None

Outputs

TICK: The current system time in ticks. This structure is defined in `SymbolTime.h`.

Notes

None

AN1066

TickGetDiff

This function returns the time difference between the two tick values.

Syntax

```
TICK TickGetDiff( TICK, TICK );
```

Inputs

None

Outputs

TICK: The current system time in ticks. This structure is defined in `SymbolTime.h`.

Notes

Both of the inputs need to be saved to local variables. Users should not implement `TickGetDiff` that uses the return value from another function (i.e., `TickGetDiff(TickGet(), oldTick)` is not valid).

WriteData

This function writes the input data to the transmit FIFO.

Syntax

```
void WriteData( BYTE );
```

Inputs

The data to be sent to the transmit FIFO.

Outputs

None

Notes

None

USER CONSIDERATIONS

There are several different network situations and circumstances that are not inherently covered by the Stack. Each of these situations should be considered. Some of the situations must be implemented. Others can be implemented if the system requires it.

Which Network to Join?

The network discovery feature built into the MiWi protocol searches the available channels for networks. It does not, however, choose which network to join. This is left as an application decision. Some applications will want to pick one network over another, or a certain coordinator over another coordinator, within the same network. After the network discovery is complete each device will then need to search through the list and determine to which coordinator it will join.

Failure Recovery

Failure recovery is an interesting issue in wireless communications. Some developers require their networks to dynamically heal when parts of the network fail. Other developers need the network to stay as unchanged as possible, even when failures do occur. Because of this variation in requirements, the MiWi protocol Stack doesn't default to either implementation.

The MiWi protocol provides ACKs on both the MAC and MiWi protocol layers. This allows users to know that their packet reached the destination correctly or to determine that the packet did not make it to the destination successfully. These ACKs can be used to help determine when there is a network failure.

Determining when communication to a node is also not a feature of this implementation. Determining a node is no longer available could be a failed packet ratio, a number of consecutive packets failed, a low RSSI, a low data throughput, etc. The current Stack does not implement any of these requirements. If an application has such requirements, then it should be added in at the application level (or change the Stack functionality, if required). To remove a node from the network table, only the `networkTable[].status.isValid` bit needs to be cleared.

One possible mode of failure is if the parent of a device fails. In some networks, it is preferred that the orphaned device find a new parent, while in other networks, the device is left off of the network until the parent returns online. Some implementations may prefer to rejoin the same network in a different location, while others may prefer to search for a new network.

Another problem that is not covered in the Stack is how to promote a node from a coordinator (or end device) to a PAN coordinator if the PAN coordinator fails. Because the Stack does not determine when a device is offline, it also can not initiate this process. Promoting a coordinator to a PAN coordinator can create issues in that the coordinator's old children must all be dropped off of the network. Remember that the PAN coordinator always has the address, 0000h. This means that the coordinator that is taking over the role must change its address, and thus, all of its previous children's addresses must change as well.

Which coordinator should take control of the network is also a decision that is better suited for the application to decide. Many networks can exist just fine without the PAN coordinator present. Both routing and end device joining work just fine without the PAN coordinator. However, no new coordinators will be allowed to join the network while the PAN coordinator is offline.

Exchanging EUIs to Protect Against Node Migration

Nodes in a wireless network may change their parent for various reasons, including failure and mobility. In this situation, the device's short address will change. Nodes that were communicating with the node that moved will no longer be able to communicate with the node. It is because of this reason that devices that care about maintaining connectivity, despite node mobility, may find it useful to request a node's EUI after establishing communications with that device. Once a node has a EUI for another node, it will be able to send out a `DiscoverNodeByEUI()` request to find the new short address for that device. The `OpenSocket()` request will provide both a short and long address (EUI) for the node when it makes the connection. A EUI only needs to be discovered for nodes that form communication links via other methods.

Accepting an Open Socket Response

The Stack provides a means of forming dynamic bonds between two devices through the `OpenSocket()` request function. When the request returns a result, this merely indicates that another device was looking for a communication partner in the same time frame. This does not imply that the devices are meant to talk to one another. Deciding if the returned node is acceptable or not is an application level decision and may require implementation if required.

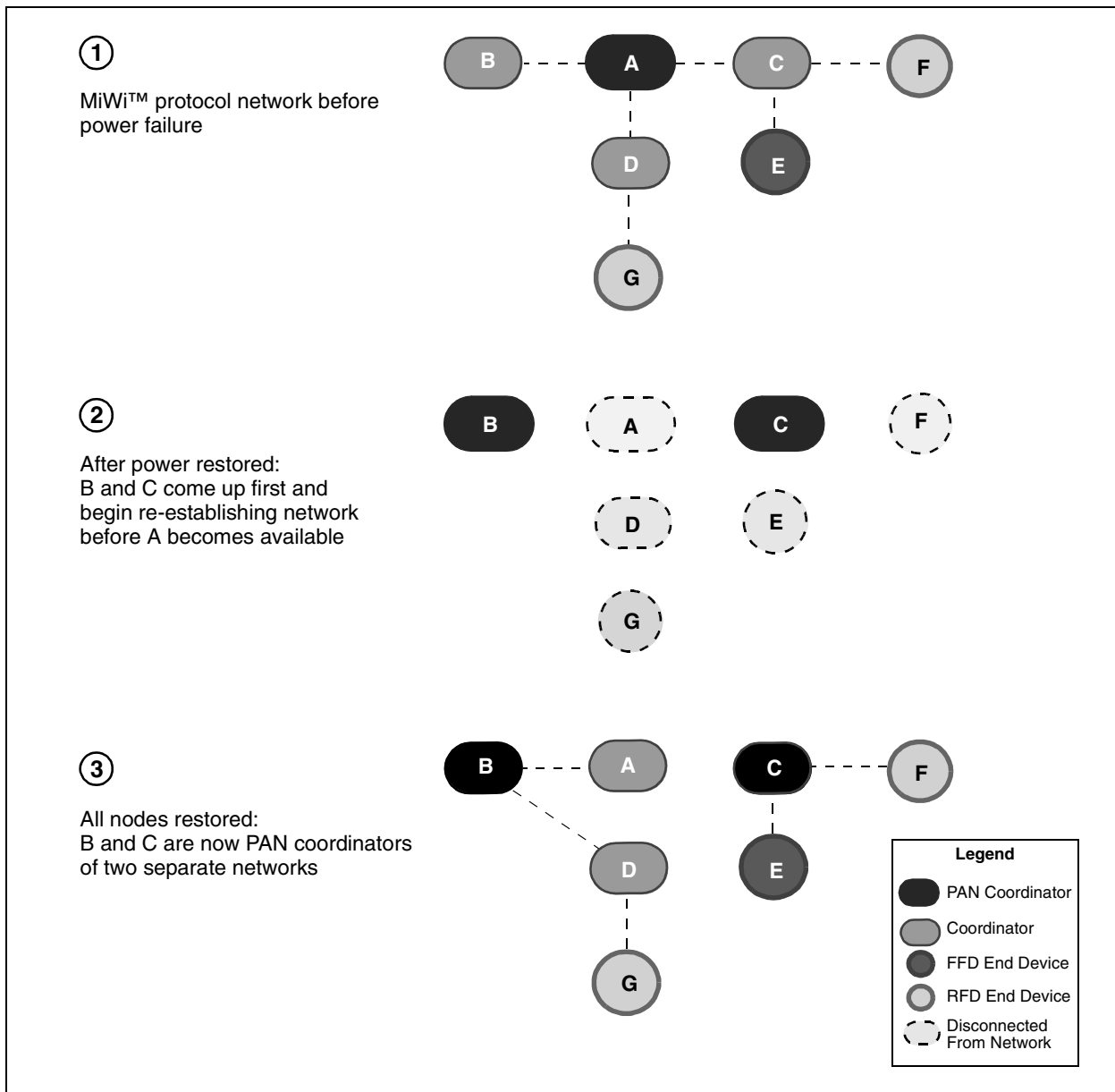
Network Islands

The coordinators and PAN coordinator in the MiWi protocol don't have much difference in terms of functionality. Because of this, it is possible to create a coordinator that will take on the role of a PAN coordinator if it can't find a suitable network. This poses an interesting problem if the entire network is power cycled. Take, for example, a network with a PAN coordinator, two coordinators located outside of each others' radio range and various end devices. If the entire network power cycles, and the two coordinators come online first, they will both search for a PAN coordinator. Neither will see a PAN coordinator and each of them will form a network of their own, possibly on different channels. Some time later, the original PAN coordinator will power

on and attempt to find a network. It will find the network formed by each of the PAN coordinators and join one of the networks. This forms two distinct networks that can no longer communicate, instead of one large network.

As an example, consider the network in Figure 12. Sequence 1 shows how the network operates when it is functioning correctly. After a power failure, coordinators B and C come back online first (sequence 2). They are unable to see each other and unable to find node A so they both start their own network. After some time, the other nodes in the network join. When node A comes back online, it will see two networks it can join to and pick one. The two networks will remain disjointed unless this is either prevented or corrected.

FIGURE 12: FORMATION OF MiWi™ PROTOCOL NETWORK ISLANDS



There are several possible solutions to the network islands problem. The first solution is to define the coordinators without their PAN coordinator capable bit set. This forces the coordinators to find a network and never form a network of their own. The downside is that while the PAN coordinator is offline, the entire network will remain broken.

Another method is to write the application so that the network forms a fixed PANID on a fixed channel. In this method the PAN coordinator comes online and detects two networks with the same PANID on the same channel and sends a PAN conflict packet indicating to these devices that there was a problem in the network topology. In this scenario, the network regains some functionality in the location of the two coordinators while the PAN coordinator is offline. The drawback to this method is that detecting the conflict can be difficult. The children of the coordinators will need to be forced off the network, then rejoin, and the nodes are now limited to one network and one PANID; this eliminates their ability to prevent network collisions with other networks.

No matter the circumstance a network will have to address this issue and determine and develop an appropriate solution for that application.

Security

There may be instances where it is necessary to transmit the key. For example: in some applications, devices may join the network without a preconfigured key and get assigned a key once on the network. Another example: some networks update their key periodically.

Users are encouraged to implement some form of key handling procedure in the application layer with a separate Report Type and Report ID. One method of doing it is to transfer the security key and key sequence number every time a node joins the network, and store the keys and key sequence number in RAM. This approach only meets the minimum requirement of a secured network, since the security key is transmitted every time a node joins the network.

An alternative approach is to set a default key for every device and use the default key to secure the transferred key. This approach provides minimum protection for the keys. A third safe approach is to add a new function to store data to the program memory during run time. This way, the security key and key sequence number only need to be transferred once when the device initially joins the network. Once obtained, the security information is retained in program memory, and does not have to be retransmitted if the device leaves and rejoins the network.

These key handling methods are suggestions only. If this functionality is required by an application, it is left to the user to implement a suitable method.

RESOURCE REQUIREMENTS

The size of the final application is determined by its device type, as defined by both IEEE 802.15.4 and the MiWi protocol, its configuration and the use of security features. Table 10 gives the program memory and RAM requirements for the basic device configuration, while Table 11 shows the incremental memory requirement for additional features. Note that these are only the requirements for implementing the MiWi protocol; users must still account for their own application's needs.

For example, a coordinator that enabled cluster sockets, P2P sockets and EUI discovery, will require approximately $9,110 + 778 + 924 + 604 = 11,416$ bytes of program memory.

Note: The memory requirements provided here reflect those of the initial release of the MiWi Wireless Networking Protocol Stack at the time of the initial publication of this document. Subsequent code revisions may increase or decrease these requirements. For the requirements of a particular release, see the `readme.txt` file included with that release's distribution.

TABLE 10: PIC18 MEMORY REQUIREMENTS FOR THE MiWi™ PROTOCOL STACK DEMO AND BASIC DEVICE CONFIGURATIONS⁽³⁾

Basic Device Configuration	MiWi Protocol Security	Program Memory (bytes)	RAM (bytes) ^(1,2)
Coordinator Capability	Not Used	9470	70 + RX Buffer Size + TX Buffer Size + (13 * Network Table Size) + Indirect Buffer Size
	Enabled	12590	70 + RX Buffer Size + (2 * TX Buffer Size) + (13 * Network Table Size) + Indirect Buffer Size
RFD End Device	Not Used	7210	50 + RX Buffer Size + TX Buffer Size + (13 * Network Table Size)
	Enabled	10478	50 + RX Buffer Size + (2 * TX Buffer Size) + (13 * Network Table Size)
FFD End Device	Not Used	6784	50 + RX Buffer Size + TX Buffer Size + (13 * Network Table Size)
	Enabled	9948	50 + RX Buffer Size + (2 * TX Buffer Size) + (13 * Network Table Size)
P2P Only Device	Not Used	3580	50 + RX Buffer Size + TX Buffer Size + (13 * Network Table Size)
	Enabled	6706	50 + RX Buffer Size + (2 * TX Buffer Size) + (13 * Network Table Size)

- Note 1:** These numbers do not include the required RAM for the C Stack.
- Note 2:** TX and RX buffers each must be at least 36 bytes. The indirect buffer must be at least 41 bytes. The network table size must be at least 1 byte.
- Note 3:** These values are from code compiled with C18 v3.04 in MPLAB® IDE v7.43.

TABLE 11: PIC18 MEMORY REQUIREMENTS FOR ADDITIONAL MiWi™ PROTOCOL STACK FEATURES⁽¹⁾

Feature Sizes	Additional Program Memory (bytes)	Additional RAM (bytes)
Cluster Sockets on Coordinator	778	0
Cluster Sockets on End Device	396	0
P2P Sockets on Coordinator	924	0
P2P Sockets on End Device	182	0
Discover Node by EUI on Coordinator	604	0
Discover Node by EUI on End device	194	0

- Note 1:** These values are from code compiled with C18 v3.04 in MPLAB® IDE v7.43.

CONCLUSION

For developers looking for an entry point into wireless networking, the MiWi protocol provides a low-cost platform to ease the development of short-range, wireless, networked applications. Based on an established IEEE standard, the MiWi protocol provides features that help make implementing a simple wireless network easier. It is not intended to address all of the scenarios and decisions required in creating a wireless solution.

The MiWi protocol is the entry point for Microchip's wireless solution based on IEEE 802.15.4. Users needing a more complex networking solution may want to consider the Microchip implementation of the ZigBee protocol or expanding the MiWi protocol to suit their needs.

REFERENCES

IEEE Std 802.15.4-2003, *Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low Rate Wireless Personal Area Networks (WPANs)*. New York: IEEE, 2006.

"ZENA™ Wireless Network Analyzer User's Guide" (DS51506) <http://www.microchip.com>

Microchip Application Note AN965, "Microchip Stack for the ZigBee™ Protocol" <http://www.microchip.com/zigbee>

"PICDEM™ Z Demonstration Kit User's Guide" (DS51524) <http://www.microchip.com>

APPENDIX A: SOURCE CODE FOR MiWi™ WIRELESS NETWORKING PROTOCOL STACK

Because of its size, a complete source code listing of the MiWi Wireless Networking Protocol Stack is not provided here. The complete source code, including the ZENA analyzer demo software and MiWi protocol demo applications, is available for download from the Microchip corporate web site, at

www.microchip.com/miwi

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Information contained in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION, INCLUDING BUT NOT LIMITED TO ITS CONDITION, QUALITY, PERFORMANCE, MERCHANTABILITY OR FITNESS FOR PURPOSE. Microchip disclaims all liability arising from this information and its use. Use of Microchip devices in life support and/or safety applications is entirely at the buyer's risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights.

Trademarks

The Microchip name and logo, the Microchip logo, Accuron, dsPIC, KEELOQ, KEELOQ logo, microID, MPLAB, PIC, PICmicro, PICSTART, PRO MATE, PowerSmart, rfPIC, and SmartShunt are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.


Amplab, FilterLab, Linear Active Thermistor, Migratable Memory, MXDEV, MXLAB, PS logo, SEEVAL, SmartSensor and The Embedded Control Solutions Company are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Analog-for-the-Digital Age, Application Maestro, CodeGuard, dsPICDEM, dsPICDEM.net, dsPICworks, ECAN, ECONOMONITOR, FanSense, FlexROM, fuzzyLAB, In-Circuit Serial Programming, ICSP, ICEPIC, Mindi, MiWi, MPASM, MPLAB Certified logo, MPLIB, MPLINK, PICkit, PICDEM, PICDEM.net, PICLAB, PICtail, PowerCal, PowerInfo, PowerMate, PowerTool, REAL ICE, rfLAB, rfPICDEM, Select Mode, Smart Serial, SmartTel, Total Endurance, UNI/O, WiperLock and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.

All other trademarks mentioned herein are property of their respective companies.

© 2007, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.

 Printed on recycled paper.

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
== ISO/TS 16949:2002 ==

Microchip received ISO/TS-16949:2002 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona, Gresham, Oregon and Mountain View, California. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.



WORLDWIDE SALES AND SERVICE

AMERICAS

Corporate Office

2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
<http://support.microchip.com>
Web Address:
www.microchip.com

Atlanta

Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Boston

Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago

Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Dallas

Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit

Farmington Hills, MI
Tel: 248-538-2250
Fax: 248-538-2260

Kokomo

Kokomo, IN
Tel: 765-864-8360
Fax: 765-864-8387

Los Angeles

Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

Santa Clara

Santa Clara, CA
Tel: 408-961-6444
Fax: 408-961-6445

Toronto

Mississauga, Ontario,
Canada
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Asia Pacific Office

Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon
Hong Kong
Tel: 852-2401-1200
Fax: 852-2401-3431

Australia - Sydney

Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing

Tel: 86-10-8528-2100
Fax: 86-10-8528-2104

China - Chengdu

Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

China - Fuzhou

Tel: 86-591-8750-3506
Fax: 86-591-8750-3521

China - Hong Kong SAR

Tel: 852-2401-1200
Fax: 852-2401-3431

China - Qingdao

Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

China - Shanghai

Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

China - Shenyang

Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

China - Shenzhen

Tel: 86-755-8203-2660
Fax: 86-755-8203-1760

China - Shunde

Tel: 86-757-2839-5507
Fax: 86-757-2839-5571

China - Wuhan

Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

China - Xian

Tel: 86-29-8833-7250
Fax: 86-29-8833-7256

ASIA/PACIFIC

India - Bangalore

Tel: 91-80-4182-8400
Fax: 91-80-4182-8422

India - New Delhi

Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

India - Pune

Tel: 91-20-2566-1512
Fax: 91-20-2566-1513

Japan - Yokohama

Tel: 81-45-471- 6166
Fax: 81-45-471-6122

Korea - Gumi

Tel: 82-54-473-4301
Fax: 82-54-473-4302

Korea - Seoul

Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

Malaysia - Penang

Tel: 60-4-646-8870
Fax: 60-4-646-5086

Philippines - Manila

Tel: 63-2-634-9065
Fax: 63-2-634-9069

Singapore

Tel: 65-6334-8870
Fax: 65-6334-8850

Taiwan - Hsin Chu

Tel: 886-3-572-9526
Fax: 886-3-572-6459

Taiwan - Kaohsiung

Tel: 886-7-536-4818
Fax: 886-7-536-4803

Taiwan - Taipei

Tel: 886-2-2500-6610
Fax: 886-2-2508-0102

Thailand - Bangkok

Tel: 66-2-694-1351
Fax: 66-2-694-1350

EUROPE

Austria - Wels

Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen

Tel: 45-4450-2828
Fax: 45-4485-2829

France - Paris

Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Munich

Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Italy - Milan

Tel: 39-0331-742611
Fax: 39-0331-466781

Netherlands - Drunen

Tel: 31-416-690399
Fax: 31-416-690340

Spain - Madrid

Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

UK - Wokingham

Tel: 44-118-921-5869
Fax: 44-118-921-5820