GT202 WiFi Module API Guide v1.3

June 2014

Contents

1 Introduction

This document provides detailed information intended for a software driver developer. The information presented can be used to enhance, extend, or adapt the reference source code to meet customer requirements. This document does not attempt to detail every subject; rather, it allows the reader an opportunity to generally understand what the various components are and how they interact. Perusal of the code may provide more depth of understanding. Header files, especially ones that describe formal APIs, include valuable comments that exhaustively describe each interface and parameter.

1.1 Overview

The GT202 WiFi module contains a wireless LAN (WLAN) module that provides IEEE 802.11 WLAN network capability including:

- QCA4002 WiFi SoC,
- Wireless Media Access Control (MAC)
- Radio
- Baseband
- IEEE 802.11 protocol processing handled by an on-chip network processor
- An integrated antenna and RF sub system
- Integrated on-board antenna, with external antenna option
- RF matching circuit
- Component required to run a WiFi communications channel

The GT202 WiFi module provides IEEE 802.11 wireless network capability to a host system (e.g., a hand-held device) with embedded processor, memory, and a standard interconnect (SPI) and achieves excellent performance with low power. The host system is not required to understand wireless protocols or radio hardware.

The GT202 WiFi module appears like an ordinary NIC card (e.g., Ethernet card), but provides high-level wireless management features for a host to control some aspects of wireless operation. The chipset network processor executes software (firmware) from ROM and RAM.

Host software can be Freescale MQX RTOS running on the Kinetis K-Series MCU or

Freescale MQX-Lite RTOS running on the Kinetis KL-series MCU..

1.2 Hardware

The QCA4002 chipset on GT202 WiFi module includes a embedded processor, DMA engines, external memory controller, control logic, a wireless MAC and baseband, a radio, general purpose I/Os (GPIOs), serial port

The host system can be Freescale Kinetis series MCU, and the host can communicate with the QCA4002 chipset target using mechanisms and protocols supported by the hardware and the firmware, the chipset can provide wireless networking to the host.

The communication between the host and the QCA4002 chipset target occurs primarily through messages sent through a mailbox (bidirectional FIFO queues) and special-purpose registers on the QCA4002. The mailboxes and the registers are accessed through special addresses mapped to the device address space and through the services provided by a standard interconnect.

Interconnect is anything that allows the host and target to communicate. To support an interconnect, the QCA4002 chipset must be embedded into a device (e.g. a board) compatible in terms of form factor, electrical characteristics, pinout, signaling, commands and functions, addressing, protocols, and so on. The host software must also use an appropriate host driver to manage the attached chipset.

1.3 Software (host and target)

The GT202 WiFi module software is partitioned into host-side software and target-side firmware. Firmware runs on the QCA 4002 chipset network processor and is stored in target memory. Host software runs on the host CPU and uses host memory.

The target-side firmware is written, owned, controlled, and maintained by Qualcomm® Atheros and is supplied in binary form only. The on-chip ROM of the QCA4002 chipset is preloaded with an OS and the WLAN firmware. The QCA4002 firmware begins execution from the ROM and executes a downloaded application that may install additional firmware from the external serial flash to instruct the RAM inside the QCA4002.

Reference host software are provided for selected platforms. To facilitate development of host software, source code for the host software reference implementation is supplied with the PDK (Platform Development Kit) distributions. System integrator may leverage the supplied host software components to create application-specific host software, or even use them without modification. While system integrators may leverage host software components, ultimately the system developer is responsible for the host software on their platforms, OSs, and interconnects.

It is expected that customers will ensure they have valid licensing in place for any third party reference software included in the development kits. If unsure, please contact the appropriate third party software provider.

1.3.1 Host software adaptation

To use the GT202 WiFi module within a host system, host software must be developed or ported from the reference source provided in the PDK. The end product does not need to look like the reference implementation. The APIs presented to the host OS, the APIs presented to users, and even the internal APIs, the architecture, and the design within the host wireless support function can be completely different as long as the host can communicate using the defined message interfaces.

The extent to which an implementation leverages the reference code determines the effort to integrate future changes (enhancements and fixes) into that implementation. To leverage as much code as possible, the OS-specific code is separated from the generic code in the reference source. The header file <processor>_customer/custom_src /include/a_osapi.h is used to define OS-specific parameters of the OS services used in the driver.

The basic design separates the host software into well-defined, layered modules with APIs (see Figure 1-1). It also facilitates code maintenance by allowing distinct code components to be replaced without a rippling effect. Some aspects of the host platform beyond the choice of OS may complicate the implementation. All data is passed across the interconnect to and from the QCA4002 in 32-bit Little Endian format.

1.3.2 Target support for host bring up

The GT202 WiFi module target provides support for the host bring-up effort. If possible, the GT202 Development Kit with K-series MCU or KL-series MCU should be used during initial host driver development. This board is a known-working board, reducing possible failure scenarios. Furthermore, it includes serial port connectors to aid in debugging. The QCA4002 serial port is used for minimal debug output.

1.3.3 Reference host software stack

Host software is organized into layers which collectively define the host software stack. In general, functions in the highest layer may call other functions at the same layer or one layer down. Functions do not make direct calls to higher layers, though upper layers may register callbacks with lower layers. The network stack is optional and can be enabled or disabled by setting the macro ENABLE_STACK_OFFLOAD in the mqx\source\io\enet\atheros_wifi\ custom_src\include\a_config.h header file. This setting must be made before BSP compilation. This macro is set to 1 by default which makes the host software use the Offloaded Stack in the QCA4002 chip.

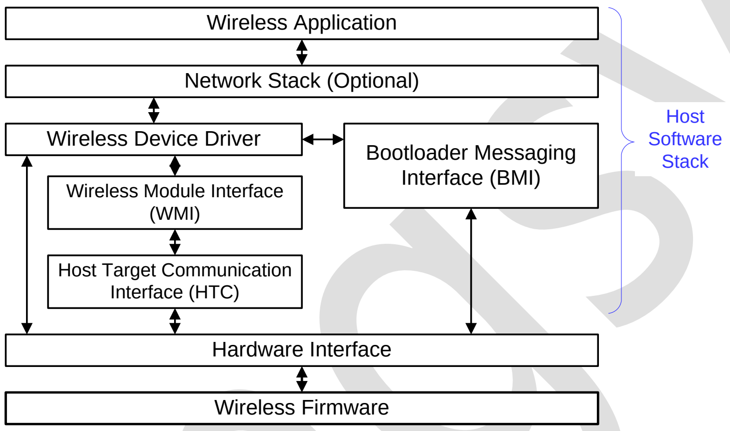

Figure 1-1 Host software layers

Table 1-1 describes these host software layers. While some names are similar to those used in the ISO/OSI network model, they have no relationship. The layers described here are part of the design of the host wireless stack, part of the data link layer in the ISO/OSI network model, or part of the data link layer in the TCP/IP network model. Descriptions apply to the Qualcomm Atheros reference host stack.

Table 1-1 Host software layers

Layer	Description
Wireless Application	The wireless application layer produces data to send over wireless and consumes incoming data from wireless. It can also produce wireless control requests (e.g. IOCTLs). This layer consists of the top of the host system's traditional network stack and calls to the wireless device driver for service. The wireless application layer is platform-specific and provides the glue between the platform-independent software and a particular platform. It handles application requests to send and receive data, and translates OS-specific control commands into platform- and interconnect-independent requests the target understands.
Network Stack	The QCA4002 contains an IP stack in the target. The driver implements a thin socket layer which provides a simple BSD socket-like API. The socket layer source is at common_src/stack_common and custom_src/stack_custom.
Wireless Device Driver/ Wireless Module Interface (WMI)	When the wireless application needs to send control messages to the chipset, it calls into the WMI layer to create the messages. This layer understands the host/target WMI protocol. The source is at common_src/wmi/. The header file include/wmi.h lists all messages from host to target (commands) and from target to host (requests and events).
Host/Target Communications (HTC)	The wireless device driver calls into the HTC layer to handle message transport. HTC does not understand the contents of messages it transports (only WMI does), but it understands the mechanics of messaging with the QCA4002chipset. This layer handles flow control and knows the chipset addresses to read and write to relay messages. The source is at common_src/htc/.

Layer	Description
Hardware Interface	The hardware interface is the software layer between the driver and the SPI hardware interface. The driver calls into this layer when it needs to access the chipset address space. This layer abstracts register and memory access details and provides an interconnect- and platform-independent API. The source is found at common_src/hw_interface. The common hardware interface layer calls into the custom hardware interface. The source is at custom_src/hw_interface.
Bootloader Message Interface (BMI)	This layer is used during firmware upgrade. The source is at common_src/bmi.

Data transmitted from the host traverses these software layers and transfers to the physical hardware interface/bus. The QCA4002 wireless module receives the data from the hardware interface and the target firmware performs additional processing and delivers the data to the air. Control messages also make their way down this stack and over the hardware interface, where the firmware interprets them and takes appropriate control actions.

When incoming data arrives over the air (or when the chipset wishes to report an event of interest), the flow is reversed: a message is created by the QCA4002 chipset target and sent over the physical hardware interface to the host. The host detects a pending message (via an interrupt) and retrieves the message through the hardware interface. Received messages are passed up the stack for processing.

1.3.4 Software Messaging

All communication between host and target is comprised of messages that contain a message header, followed by payload data specific to the command. The target firmware and host software share several messaging interface specifications defined by a set of subcomponents (i.e., WMI, and HTC). The message interfaces are defined in a header file for that interface. Messages may be transmitted between the host and target through a message transport (e.g. HTC). The message transport handles flow control mechanisms to ensure reliable delivery, and the required framing/padding. Table 1-2 describes the main message interfaces. All of these message interfaces use 32-bit little-Endian byte ordering.

Table 1-2 Message interface

Interface	Description
Wireless Module Interface (WMI)	WMI is used during normal operation to: ▪ Transmit data over wireless; ▪ Receive data from wireless; ▪ Control wireless operation. To control wireless operation, the host sends high-level WMI commands to the target. The target replies to some commands and asynchronously notifies the host of WMI events. Flow control and framing for WMI is handled by the HTC message transport using a credit system.
	Example WMI command messages (host-to-target) ▪ CONNECT (e.g. to an AP) ▪ START_SCAN (e.g. for suitable APs) ▪ GET_STATISTICS
	Example WMI event messages (target-to-host) ▪ WMI_READY ▪ DISCONNECT (e.g. from an AP) ▪ REPORT_STATISTICS (e.g. for an earlier request from the host)

1.4 Startup sequence

This section describes the startup steps from power up until wireless networking enabled. Startup proceeds through a few phases on both host and target. Table 1-3 presents these phases in chronological order with descriptions of host and target activities.

Table 1-3 Startup sequence phases

Phase	Description
Initialization	IO Enable: Once the host driver is loaded, it enables the QCA4002 wireless module. Once the target is enabled, it executes firmware that initializes SoC and software states. QCA4002 wireless components remain disabled and irrelevant during this period. Once initializations are complete, the target indicates to the host that it is ready.

Phase	Description
Bootstrap	The boot sequence enters a stage which interacts with the host code. The target enters this stage after the initialization. During this time the host writes to a known area to indicate to the target whether to proceed executing the regular image on the firmware NVRAM or to download code from the host. If a new firmware is not being downloaded, the host then waits for the target to indicate that it is ready for WMI communications.
Firmware	At the end of the bootstrap stage, the target handles a few more essential initializations. Notably, it initializes clocking so that the QCA4002 runs at the desired speed. The target then jumps to a designated target application address. The default firmware component is athwlan, the WLAN firmware by Qualcomm Atheros, which resides in ROM and/or flash. During a flash upgrade, the host loads the target flash application into target RAM and tell the target to execute that rather than athwlan. The initialization sequence in athwlan includes: target-side wireless DMA buffers → all wireless hardware → WMI communications → HTC communications. When HTC is initialized, it sends an interrupt to the host. The host then initializes host-side HTC and WMI. At this point, a user can use the various host applications to start wireless communication.

1.5 Firmware

Table 1-4 describes the firmware used in the QCA4002 system.

Table 1-4 Firmware

Application	Description
athwlan	The target-side firmware embodies the WLAN driver. Without this, the chipset has no understanding of IEEE 802.11. This target firmware is embedded in ROM and/or NVRAM. Initialization of NVRAM (off chip serial flash) is typically performed during manufacturing. A reflashing utility is provided with every release to enable firmware upgrading by storing the new firmware in the NVRAM of the module.

1.6 Configuration/customization methods

Table 1-5 describes configuration/customization mechanisms.

Table 1-5 Configuration/customization mechanisms

Mechanism	Description
WMI	WMI is used during wireless LAN operation to configure the wireless interface including: ▪ Foreground and background scan parameters ▪ Filters ▪ Wireless mode (802.11 a/b/g/n) ▪ Wireless channels/frequencies ▪ SSID probe parameters ▪ Wireless authentication, encryption, and TKIP countermeasures ▪ Association information ▪ Ad hoc beacon interval, listen interval, disconnect interval ▪ Power management parameters ▪ Tx power level ▪ Prioritized data
Bootloader	The boot loader in the ROM is used during the QCA4002 chipset startup. The boot loader interacts with host during the target boot phase of initialization. Once that boot phase is complete, the boot loader cannot be used again until the target is reset. See 1.4 Startup sequence for more information.
Target Firmware	After the bootstrap phase is complete, the QCA4002 chipset executes the wireless LAN (athwlan) firmware.
Interest Area	An area in target RAM for simple configuration items, status, and software switches. This area is cleared by firmware at startup. The host may choose to write special values to override settings, then firmware uses those values during operation.

2 IP Stack Offload

This chapter describes the offloaded IP stack APIs as well as the required host driver changes to support the offloaded IP stack. The host driver exposes a set of APIs to application developers to create TCP/IP and/or UDP/IP sockets for data transmission and reception. The IP stack offload feature supports both IP version 4 and version 6.

Driver changes are highly configurable. The compile time options (macros defined in a_config.h) allow user to enable or disable the IP stack offload feature as well as various configuration options and features. Certain features may have a significant impact on memory requirements and hence will be made optional with the compile time macros.

Table 2-1 Source organization

Common Source	This section includes all system independent functionality, including the WMI layer modifications, socket API implementation, fragmentation/reassembly, and so on.
Custom source	This section includes OS/platform-specific code changes, for example packet send and receive APIs depending on the system-specific packet structures. Similarly, blocking functionality implementation is dependent on the OS features.

2.1 Control path

All control messages are sent using WMI commands from host to device and WMI events from device to host. All user-initiated socket calls are mapped into new WMI commands that map onto the IP offload on the target SoC. The target socket layer registers a new dispatch table with the WMI module so that all socket related WMI commands can be sent directly to it. The target socket layer then interprets the WMI command and calls the corresponding socket command.

Most of these commands require a response from the stack. The response (for example socket handle and error code) are conveyed to the host using WMI events.

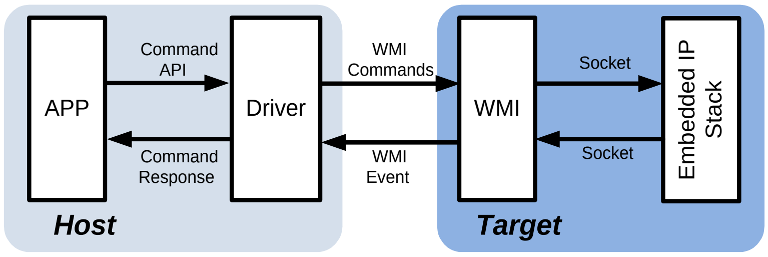

Figure 2-1 Control Path

2.1.1 WMI command implementation

The generic WMI socket command WMI_SOCKET_CMD sends socket commands to the target. The corresponding structure contains a command type field used to identify the type of the socket command.

```
typedef PREPACK struct {
 A_UINT32 cmd_type FIELD_PACKED; //Socket command type
 A_UINT32 length FIELD_PACKED;   //Number of bytes of data to follow
 A_UINT8 data[1] FIELD_PACKED;   //Start of data
} POSTPACK WMI_SOCKET_CMD;
```

Similarly, the generic WMI socket response WMI_SOCKET_RESPONSE_EVENT carries responses from the stack. A response type field indicates the type of command for which response is received.

```
typedef PREPACK struct {
 A_UINT32 resp_type FIELD_PACKED;   //Socket command response type
 A_UINT32 sock_handle FIELD_PACKED; //Socket handle
 A_UINT32 error FIELD_PACKED;        //Return value
 A_UINT8 data[1] FIELD_PACKED;     //Socket command data e.g. select FDs
} POSTPACK WMI_SOCK_RESPONSE_EVENT;
```

Following is the data structure associated with WMI_SOCKET_CMD. The data field contains the elements pertaining to the socket command. The various socket sub commands and responses can be of these types:

```
enum SOCKET_CMDS
{
  SOCK_OPEN = 0,         /*Open a socket*/
  SOCK_CLOSE,       /*Close existing socket*/
  SOCK_CONNECT,          /*Connect to a peer*/
  SOCK_BIND,        /*Bind to interface*/
  SOCK_LISTEN,      /*Listen on socket*/
  SOCK_ACCEPT,      /*Accept incoming connection*/
  SOCK_SETSOCKOPT,  /*Set specified socket option*/
  SOCK_GETSOCKOPT,  /*Get socket option*/
  SOCK_IPCONFIG,         /*Set static IP information, or get current IP cfg
*/
 SOCK_PING,         /*Send IPv4 ping*/
SOCK_STACK_INIT,    /*Initialize stack*/
 SOCK_STACK_MISC,   /*Exchange misc. info, e.g. reassembly*/
SOCK_PING6,         /*Send ICMPv6 ping message*/
SOCK_IP6CONFIG,     /*Set static IP information or get current IP cfg */
SOCK_IPCONFIG_DHCP_POOL,          /*Create DHCP pool for IPv4 */
SOCK_IP6CONFIG_ROUTER_PREFIX,     /*Create Router prefix used in Router
Advertisement */
SOCK_IP_SET_TCP_EXP_BACKOFF_RETRY,/*Configure the no of TCP retries */
SOCK_IP_SET_IP6_STATUS,       /*Enable and disable IPv6 */
SOCK_IPCONFIG_DHCP_RELEASE,  /*Release DHCP Address */
SOCK_IP_SET_TCP_RX_BUF, /*Configure Rx buffers for a TCP connection */
SOCK_HTTP_SERVER,       /*HTTP Server Command */
SOCK_HTTP_SERVER_CMD,   /*Get and post data */
SOCK_DNC_CMD,           /*Commands to reslated to resolver */
SOCK_DNC_ENABLE,        /*Enable/disable DNS Client */
SOCK_DNS_SRVR_CFG_ADDR, /*Configure DNS Server Address */
SOCK_HTTPC,             /*HTTP Client Commands */
SOCK_DNS_LOCAL_DOMAIN,  /*Configure Local Domain */
SOCK_IP_HOST_NAME,      /*Configure the local Host Name */
SOCK_IP_DNS             /*Configure DNS Database */
SOCK_IP_SNTP_SRVR_ADDR, /*Configures the sntp server addr */
SOCK_IP_SNTP_GET_TIME,  /*GET UTC Time from SNTP Client */
SOCK_IP_SNTP_GET_TIME_OF_DAY, /*Get time of day (secs)*/
SOCK_IP_SNTP_CONFIG_TIMEZONE_DSE, /*Modify time zone and enable/disable DSE
*/
SOCK_IP_SNTP_QUERY_SNTP_ADDRESS,  /*Query SNTP Server Address*/
SOCK_IP_SNTP_CLIENT_ENABLE,      /*Enable/disable SNTP client */
SOCK_SSL_CTX_NEW,    /*Create a new SSL context */
SOCK_SSL_CTX_FREE,   /*Free/close SSL context */
SOCK_SSL_NEW,       /*Create new SSL connection object/instance */
SOCK_SSL_SET_FD,     /*Add socket handle to a SSL connection */
SOCK_SSL_ACCEPT,     /*Accept SSL connection request from SSL client */
SOCK_SSL_CONNECT,    /*Establish SSL connection from SSL client to SSL server
*/
SOCK_SSL_SHUTDOWN,   /*Shutdown/close SSL connection */
SOCK_SSL_ADD_CERT,   /*Add a certificate to SSL context */
SOCK_SSL_STORE_CERT, /*Store a certificate or CA list file to flash */
SOCK_SSL_LOAD_CERT,  /*Read a cert or CA list from FLASH and add it to SSL
context */
SOCK_SSL_CONFIGURE,  /*Configure a SSL connection */
SOCK_SSL_LIST_CERT/*Request the names of the cert stored in flash*/
};
```

2.1.2 Data path uplink (host to device)

The host driver allocates headroom for IP +TCP/UDP headers before sending data to the device. The driver is aware of the type of socket (TCP vs. UDP, v4 vs. v6) because the information is stored in the socket context to allow the driver to allocate correct header space.

The space allocated for TCP/UDP IP headers is used (overloaded) to send a custom header containing the following information:

```
typedef struct sock_send {
uint_32 handle;         //Socket handle
uint_16 length;         //Payload length
uint_32 flags;     //Send flags
SOCKADDR_T name;        //IPv4 destination socket information
uint_16 socklength;
}SOCK_SEND_T;
/*For IPv6 sockets*/
typedef struct sock_send6 {
uint_32 handle;         //Socket handle
uint_16 length;         //Payload length
uint_32 flags;     //Send flags
SOCKADDR_6_T name6 //IPv6 destination socket information
uint_16 socklength;
}SOCK_SEND6_T;
```

Two custom headers are defined for IPv4 and IPv6 respectively. Depending on the type of socket, a corresponding custom header is added. IPv6 custom header is longer than IPv4 header and thus only fits in IPv6 header space.

Custom headers are required in scenarios where multiple sockets are transmitting data. The above information is used by the firmware to populate the parameters of TCP/UDP IP stack send APIs.

WMI header modification

The WMI header will be modified to indicate a stack offload data packet. This will allow the same target firmware image to work with raw transmissions (required for a target based IP stack offload) as well as IP packets (for a host based IP stack). A new META version will be defined for packets destined for IP offload target stack that will be reflected in the EMI header portion.

```
#define WMI_META_VERSION_5 (0x05) //Used for stack offload packets
typedef PREPACK struct {
A_UINT8 reserved FIELD_PACKED;
} POSTPACK WMI_TX_META_V5;
```

MAC header

The MAC header can only be populated after ARP exchange is completed (to obtain destination MAC address). The driver allocates space for MAC header, which needs to be filled by firmware.

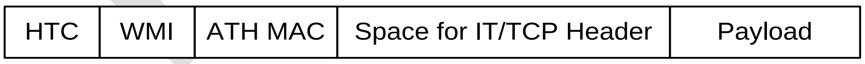

Figure 2-2 Packet structure

Figure 2-2 depicts the packet structure for packets sent from the host to the target. The TCP/UDP IP header space varies depending upon the type of socket and protocol and is dynamically allocated. A custom header will be added at the beginning of the TCP/UDP IP header space.

Custom alloc/free API

Uplink packet transmission requires supporting infrastructure (e.g. headroom for TCP/IP, netbuf etc.). In memory constrained systems, resources may be temporarily unavailable which may lead to allocation failure during a SEND operation. To ensure success of SEND operation, a CUSTOM_ALLOC API is provided which must be called prior to transmit to obtain a transmit buffer. This API will ensure that enough resources are available to transmit the requested number

of bytes. It will return an error if resources are unavailable. In case of success, it will return a pointer to a buffer of requested size.

If the buffer is no longer needed, the application must invoke CUSTOM_FREE to free the buffer. This API will ensure that all relevant allocations are freed in the correct sequence. It is important to note that calling legacy A_FREE API on a TX buffer will lead to memory leaks and must be avoided. For reference implementation, please look at ath_udp_echo() function in **throughput.c** in the throughput_demo example project.

Downlink (target device > host)

The target firmware/IP stack receives packets from RF, processes them, and forwards them to the host. No explicit receive call is sent to the target. Modifications must be made to the stack to asynchronously push data to the host. The asynchronous push is necessary for improved performance as it removes the latency involved with pull mechanism.

The firmware strips off TCP/UDP IP headers and adds a custom header before sending packets to the host, allowing the driver to identify the destination for incoming packets (especially useful in case of multiple open sockets). The custom header contains these elements:

```
typedef PREPACK struct sock_recv {

uint_32 handle FIELD_PACKED; //Socket handle
SOCKADDR_T name FIELD_PACKED;     //IPv4 destination socket information
uint_16 socklength FIELD_PACKED;  // Length of sockaddr structure
uint_32 reassembly_info FIELD_PACKED;  //Placeholder for reassembly
}POSTPACK SOCK_RECV_T;
typedef PREPACK struct sock_recv6 {
 uint_32 handle FIELD_PACKED;     //Socket handle
SOCKADDR_6_T name6 FIELD_PACKED; //IPv6 destination socket info
uint_16 socklength FIELD_PACKED;  // Length of sockaddr structure
uint_32 reassembly_info FIELD_PACKED; //Placeholder for reassembly
}POSTPACK SOCK_RECV6_T;
```

Firmware must pick the correct structure depending upon the type of socket, thus allowing optimized space use and reducing the latency of transmitting over HTC. On the host, the driver buffers the incoming packets until the application is ready to receive the data (the application calls recv/recvfrom). The number of receive buffers in the driver may be limited.

Packet buffering mechanism

Incoming packets are placed in a socket-specific queue using the socket handle available in the packet's custom header. Head and tail pointers in the socket context provide a simple mechanism for queue traversal. Packets are pulled from the head.

When the application calls the packet receive API, the driver checks the queue for pending packets. If a packet is available, the receive call returns the packet immediately. If the packet is not available, the application may block if it made a blocking receive call. If the receive call is non-blocking, the receive API returns the value -1. When the driver places incoming packets in the socket queue, it checks whether the application is blocked on receive. If so, it unblocks the application allowing it to de-queue the packet.

Freeing the buffer

Data can pass to the application in two ways: doing a copy to application buffer or zero-copy (passing a pointer to buffer). A ZERO_COPY macro is used to switch between the mechanisms.

- Buffer allocated/freed by the application

In this traditional approach, the application allocates a buffer and passes a pointer in the receive call. The driver does a copy of a pending packet from the queue and frees its own buffer. The application processes the incoming packet. It may then decide to reuse that buffer or free the application buffer. The obvious advantages to this method are:

  - demarcation between application and drive The application developer manages its own buffers, providing a clear r.
  - This is a familiar process, as no new callback functions are needed.

  The disadvantages to this scheme are:

  - The extra copies affect performance.
  - More resources are needed and this may not be feasible in a low resource systems.

- Zero-Copy

The application does not allocate any buffers. It passes a double pointer in the receive call. On return, the pointer will point to driver's own Rx buffer. The driver removes the buffer from the socket queue and places it in a common To-Be-Freed queue. Once the application is done processing the packet, it calls a zero_copy_free API. This API will searches the common queue for the packet and frees it.

This more efficient design does not require the application to allocate any buffers and buffer copy overhead is avoided. However, it is imperative that the application calls the zero_copy_free API, or the driver runs out of Rx buffers leading to deadlock situation.

2.1.3 Design considerations

Compile time options

These compile-time macros can be used to enable or disable features in the IP stack. The macros allow for optimizing the driver for memory constrained host platforms.

Table 2-2 Compile Time Options

Option	Defined In	Purpose
ENABLE_STACK_OFFLOAD	**a_config.h**	Enable or disable the stack offload feature. Useful when TCP/UDP IP stack is on the host.
MAX_SOCKETS_SUPPORTED	**atheros_stack_offload.h**	Limit the maximum number of sockets that can be supported.
ZERO_COPY	**atheros_stack_offload.h**	The macro enables the zero copy feature. In memory-constrained systems, the application provides a pointer instead of allocating a buffer to receive incoming data. The driver returns the pointer to received packet instead of copying the packet to the application buffer. The application must call the zero_copy_free() API after it is done with the buffer and pass the pointer to the buffer.
NON_BLOCKING_TX	**atheros_stack_offload.h**	If this macro is enabled, the application thread returns after submitting packet to the driver thread. Under this setting, the application must not try to reuse or free this buffer. This buffer is now owned by the driver. If the application needs to send another packet, it should call CUSTOM_ALLOC again to get a new buffer.

Option	Defined In	Purpose
NON_BLOCKING_RX	**atheros_stack_offload.h**	Prevents a blocking receive operation. Application thread checks the Rx queue and returns immediately.

2.1.4 Socket context

A socket context structure is used to track all open sockets. Because the number of sockets is limited at compilation, the corresponding number of contexts is allocated during initialization phase. A socket context consists of a common context and a custom context. The custom context can be accessed via the common context. The common portion of the context includes:

- Socket handle
- Socket type (v4/v6)
- Protocol (TCP/UDP/raw)
- Some socket options

The custom portion includes:

- OS-dependent construct to enable blocking
- Receive queue (To hold incoming packets)
- Flag to indicate blocking state
- Flag to indicate availability of response from target

An empty socket context is identified by a zero socket handle value. When a socket is created, the socket handle is populated thereby indicating that the context is now in use. When a socket is closed, the structure is zeroed out, freeing the context. Socket handle is used to fetch the corresponding common context. All incoming data packets and WMI events include the socket handle which is then used to obtain the context.

2.1.5 Multiple socket support

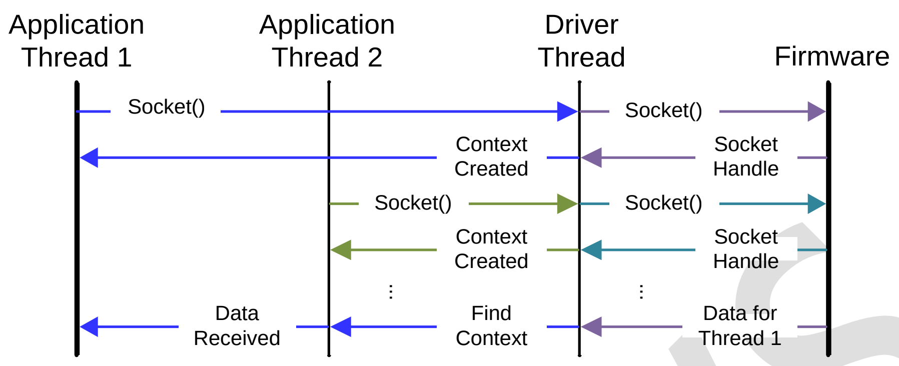

Figure 2-3 Multiple Concurrent Sockets

The socket handle will be used to distinguish the socket to which data will be sent or received. All packets received from the target will contain a custom header with the socket ID to allow the driver to route the packet to the correct socket.

2.1.6 Blocking vs. non-blocking sockets

To achieve blocking functionality, a semaphore blocks the application thread. The semaphore is released under two conditions; when a packet is received for that socket, or if a timeout occurs.

Other application threads may continue to use their sockets as the driver thread will not be blocked.

2.2 Supported APIs

This section describes a set of socket APIs provided o application developers that resemble standard socket APIs as closely as possible. Each API maps to a WMI command by the driver, which is then translated to the corresponding socket API by the WMI module on the target.

API	Description
2.2.1 t_socket()	Open/create a socket.
2.2.2 t_shutdown()	Close an existing socket.
	Connect to another socket.
2.2.4 t_bind()	Assign a local socket address to an unnamed socket

API	Description
2.2.5 t_listen()	Listen for incoming connections.
2.2.6 t_accept()	Accept incoming connections on an open socket.
2.2.7 t_select()	Allow an application thread to block on a given socket handle for a specified time period.
2.2.8 t_setsockopt()	Set options for an existing socket.
2.2.9 t_getsockopt()	Get options from an existing socket.
2.2.10 t_sendto()	Send data on a datagram socket.
2.2.11 t_send()	Send data on a stream socket which is in connected state.
2.2.12 t_recvfrom()	Receive data on a datagram socket.
2.2.13 t_recv()	Receive data on a stream socket which is in connected state.
2.2.14 t_ipconfig()	Set/get IPv4 parameters.
2.2.15 t_ip6config()	Get IPv6 parameters.
2.2.16 t_ping()	Send an IPv4 ping.
2.2.17 t_ping6()	Send an IPv6 ping.
2.2.18 t_ipconfig_dhcp_pool()	Create DHCP pool.
2.2.19 t_ip6_config_router_prefix()	Configure IPv6 prefix for Router Advertisement.
2.2.20 custom_ipconfig_set_tcp_exponential_backoff_retry()	Configure the number of TCP retries. Exponential backoff is applied between two successive retries.
2.2.21 custom_ipconfig_set_ip6_status()	Enable/disable IPv6 module at run time.
2.2.22 custom_ipconfig_dhcp_release	Release DHCP address.
2.2.23 custom_ipconfig_set_tcp_rx_buffer	Configure the number of Rx buffers to receive packets for a downlink TCP connection.
2.2.24 custom_ip_http_server	Start/stop the HTTP server.
2.2.25 custom_ip_httpc_method	Configure HTTP Client.
2.2.26 custom_ip_http_server_method	Get/post the values from/to the HTTP server.
2.2.27 custom_ip_dns_client	Start/stop DNS client.

API	Description
2.2.28 custom_ip_resolve_hostname	Resolve the hostname.
2.2.29 custom_ip_dns_server_addr	Configure IPv4/IPv6 DNS server address.
2.2.30 custom_ip_dns_local_domain	Configure the domain name for the device.
2.2.31 custom_ip_hostname	Configure the host name for the device.
2.2.32 custom_ip_dns	Configure DNS server database.
2.2.33 custom_ip_sntp_srvr_addr	Configure SNTP Server address.
2.2.34 custom_ip_sntp_get_time	Get SNTP time.
2.2.35 custom_ip_sntp_get_time_of_day	Get SNTP time of day in seconds and milliseconds.
2.2.36 custom_ip_sntp_modify_zone_dse	Configure SNTP time zone and enable/disable day light saving.
2.2.37 custom_ip_sntp_query_srvr_address	Query the SNTP address.
2.2.38 custom_ip_sntp_client	Enable/disable SNTP at run time.
2.2.39 SSL_ctx_new	Creates SSL context.
2.2.40 SSL_ctx_free	Free SSL context.
2.2.41 SSL_new	Create SSL connection object.
2.2.42 SSL_set_fd	Attach TCP socket descriptor to SSL connection.
2.2.43 SSL_accept	Initiate SSL handshake (when acting as server).
2.2.44 SSL_connect	Initiate SSL handshake (when acting as client).
2.2.45 SSL_shutdown	Close the SSL connection (when acting as client).
2.2.46 SL_configure	Configure SSL connection.
2.2.47 SSL_setCaList	Set a CA list.
2.2.48 SSL_addCert	Add a certificate to the SSL object.
2.2.49 SSL_storeCert	Store a certificate or CA list to flash.
2.2.48 SSL_addCert	Load a certificate or CA list from flash and store it in the SSL object.
2.2.51 SSL_**read**	Read data over SSL connection.
2.2.52 SSL_write	Write data over SSL connection.

2.2.1 t_socket()

Prototype	A_INT32 t_socket (	
	A_VOID handle,*	Driver context
	A_UINT32 domain,	Protocol Family (e.g. AF_INET, AF_INET6)
	A_UINT32 type,	Socket Type (DGRAM or STREAM)
	A_UINT32 protocol	Protocol (usually 0)
	)	
Description	Open/create a socket. This call blocks until a socket handle is returned. If no response is received or socket creation fails, error code is returned.	
Return Value	Socket handle on success, -1 on failure	

2.2.2 t_shutdown()

Prototype	A_INT32 t_shutdown(	
	A_VOID handle,*	Driver context
	A_UINT32 sockHandle	Socket handle
	)	
Description	Close an existing socket. If socket does not exist, error code is returned.	
Return Value	0 on success, -1 on failure	

2.2.3 t_connect()

Prototype	A_INT32 t_connect(	
	A_VOID handle,*	Driver context
	A_UINT32 sockHandle,	Socket handle
	A_VOID name,*	Sockaddr structure indicating peer information
	A_UINT16 length	Length of sockaddr structure
	)	

Description	Connect to another socket. If the socket type is UDP, this call specifies: ▪ The peer with which the socket is to be associated; ▪ The address to which datagrams are sent; ▪ The only address from which datagrams are received. If the socket type is TCP, this call attempts making a connection to another socket. This call blocks until a connection is established or an error is returned.
Return Value	0 on success, -1 on failure

2.2.4 t_bind()

Prototype	A_INT32 t_bind(	
	A_VOID handle,*	Driver context
	A_UINT32 sockHandle,	Socket handle
	A_VOID name,*	Sockaddr structure indicating local address to bind
	A_UINT16 length	Length on sockaddr structure
	)	
Description	Assign a local socket address to an unnamed socket. The address is provided in the name field in form of the sockaddr (or sockaddr_6) structure.	
Return Value	0 on success, -1 on failure.	

2.2.5 t_listen()

Prototype	A_INT32 t_listen(	
	A_VOID handle,*	Driver context
	A_UINT32 sockHandle,	Socket handle
	A_UINT32 backlog	Maximum length of pending connections
	)	

Description	Listen for incoming connections. When a connection request arrives, a t_accept() call is used to accept it. The t_listen is only used with socket type SOCK_STREAM. The backlog parameter defines the maximum length for the queue of pending connections (rather than maximum open connections). If a connection request arrives while the queue is full, error code ECONNREFUSED is returned to the client.
Return Value	0 on success, -1 on failure

2.2.6 t_accept()

Prototype	A_INT32 t_accept(	
	`A_VOID* handle,`	Driver context
	`A_UINT32 sockHandle,`	Socket handle
	`A_VOID* name,`	Sockaddr structure indicating local address to bind
	`A_UINT16 length`	Length on sockaddr structure
	`)`	
Description	Accept incoming connections on an open socket. This API is used after the socket is created and bound to an address, and success is returned from t_listen(). It extracts the first connection from a queue of pending connections. The t_accpet()is only used with socket type SOCK_STREAM. The call blocks if no connection is present and the socket is not non-blocking.	
Return Value	Non-negative socket descriptor on success, A_ERROR on error	

2.2.7 t_select()

Prototype	A_INT32 t_select(	
	`A_VOID* handle,`	Driver context
	`A_UINT32 sockHandle,`	Socket handle
	`A_UINT32 tv`	Block time in ms

	)	
Description	Allow an application thread to block on a given socket handle for a specified time period. This API checks for any activity on specified socket, for example arrival of a packet at the receive queue.	
Return Value	▪ A_SOCK_INVALID: Socket does not exist, or socket closed by peer ▪ A_OK: Activity detected (packet available) ▪ A_ERROR: Timeout occurred, no activity	

2.2.8 t_setsockopt()

Prototype	A_INT32 t_setsockopt(	
	A_VOID handle,*	Driver context
	A_UINT32 sockHandle,	Socket handle
	A_UINT32 level,	Option level
	A_UINT32 optname,	Option name
	A_UINT8 optval,*	Option value
	A_UINT32 optlen	Option length
	)	
Description	Set options for an existing socket.	
Return Value	0 on success, -1 on failure	

2.2.9 t_getsockopt()

Prototype	A_INT32 t_getsockopt(	
	A_VOID handle,*	Driver context
	A_UINT32 sockHandle,	Socket handle
	A_UINT32 level,	Option level
	A_UINT32 optname,	Option name

	A_UINT8* optval,	Option value
	A_UINT32 optlen	Option length
	}	
Description	Get options from an existing socket.	
Return Value	0 on success, -1 on failure	

2.2.10 t_sendto()

Prototype	A_INT32 t_sendto(	
	A_VOID* handle,	Driver context
	A_UINT32 sockHandle,	Socket handle
	A_UINT8* buffer,	Pointer to data
	A_UINT32 length,	Payload length
	A_UINT32 flags,	Send flags
	A_VOID* name,	Sockaddr structure (v4 or v6)
	A_UINT32 socklength	Length of sockaddr
	}	
Description	Send data on a datagram socket. The destination address must be specified in the name parameter. The API works for both IPv4 and IPv6 using the **sockaddr** and **sockaddr_6** structure respectively in the name parameter.	
Return Value	Number of bytes transmitted to the target on success	

2.2.11 t_send()

Prototype	A_INT32 t_sendto(	
	A_VOID* handle,	Driver context
	A_UINT32 sockHandle,	Socket handle
	A_UINT8* buffer,	Pointer to data
	A_UINT32 length,	Payload length
	A_UINT32 flags,	Send flags
	}	

Description	Send data on a stream socket which is in connected state. The API works for both IPv4 and IPv6.
Return Value	Number of bytes transmitted to the target on success

2.2.12 t_recvfrom()

Prototype	A_INT32 t_recvfrom(	
	A_VOID handle,*	Driver context
	A_UINT32 sockHandle,	Socket handle
	A_UINT8 buffer,*	Pointer to data
	A_UINT32 length,	Payload length
	A_UINT32 flags,	Send flags
	A_VOID name,*	Sockaddr structure (v4 or v6)
	A_UINT32 socklength	Length of sockaddr
	)	
Description	Receive data on a datagram socket. The socket blocks if no packet is available, unless it is designated as non-blocking (in which case it returns immediately).	
Return Value	Number of bytes received on success, -1 on error.	

2.2.13 t_recv()

Prototype	A_INT32 t_recv(	
	A_VOID handle,*	Driver context
	A_UINT32 sockHandle,	Socket handle
	A_UINT8 buffer,*	Pointer to data
	A_UINT32 length,	Payload length
	A_UINT32 flags,	Send flags
	}	
Description	Receive data on a stream socket which is in connected state. The socket blocks if no packet is available, unless it is designated as non-blocking (in which case it returns immediately).	
Return Value	Number of bytes received on success, -1 on error.	

2.2.14 t_ipconfig()

Prototype	A_INT32 t_ipconfig (	
	A_VOID handle,*	Driver context
	A_UINT32 mode,	Configuration mode
	A_UINT32 ipv4_addr,*	Pointer to IPv4 address
	A_UINT32 subnetMask,*	Pointer to IPv4 subnet mask
	A_UINT32 gateway4*	Pointer to IPv4 gateway address
	}	

Description	Set/get IPv4 parameters.
	The mode parameter can take the following values:
	`enum IPCONFIG_MODE`
	`{`
	`IPCFG_QUERY=0, //Retrieve IP parameters`
	`IPCFG_STATIC, //Set static IP parameters`
	`IPCFG_DHCP //Run DHCP client`
	`}`	

2.2.15 t_ip6config()

Prototype	A_INT32 t_ip6config (	
	A_VOID handle,*	Driver context
	A_UINT32 mode,	Configuration mode
	*IP6_ADDR_T *v6Global,*	IPv6 global address
	*IP6_ADDR_T *v6Local,*	IPv6 local address
	*IP6_ADDR_T *v6DefGw,*	IPv6 default gateway
	*A_INT32 *LinkPrefix,*	IPv6 link prefix
	*A_INT32 *GlbPrefix,*	IPv6 global prefix
	*A_INT32 *DefGwPrefix*	IPv6 default gateway prefix
	}	
Definition	Get IPv6 parameters. For IPv6 addressing, only the IPCFG_QUERY mode option is supported.	

2.2.16 t_ping()

Prototype	`A_INT32 t_ping (`	
	`A_VOID* handle,`	Driver context
	`A_UINT32 ipv4_addr`	IPv4 Destination address
	`}`	
Definition	Send an IPv4 ping. The host blocks until a result is received from target.	
Return Value	A_OK on success, A_ERROR on failure	

2.2.17 t_ping6()

Prototype	A_INT32 t_ping6 (	
	`A_VOID* handle,`	Driver context
	`A_UINT8 *ip6addr`	Pointer to IPv6 Destination address
	`}`	
Definition	Send an IPv6 ping.	

2.2.18 t_ipconfig_dhcp_pool()

Prototype	A_INT32 t_ipconfig_dhcp_pool(	
	`A_VOID* handle,`	Driver context
	`A_UINT32 *start_ipv4_addr,`	Pointer to start address of the Pool
	`A_INT32 leasetime,`	Lease time for the address
	`)`	
Description	Create DHCP pool. This API is used by DHCP Server.	
Return Value	0 on success, -1 on failure.	

2.2.19 t_ip6_config_router_prefix()

Prototype	A_INT32 t_ip6_config_router_prefix(	
	`A_VOID* handle,`	Driver context

	*A_UINT8 *v6addr,*	Pointer to ipv6 prefix
	A_INT32 prefixlen,	Length of the prefix
	A_INT32 prefix_lifetime	Lifetime of the prefix in seconds
	A_INT32 valid_lifetime	Valid life time of the prefix in seconds
	)	
Description	Configure IPv6 prefix for Router Advertisement.	
Return Value	0 on success, -1 on failure.	
Description	Configure the number of TCP retries. Exponential backoff is applied between two successive retries. Default is 11 (time out in 9 minutes). To reduce the retransmission timeout value, reduce the number of retries.	
Return Value	0 on success, -1 on failure.	

2.2.20 custom_ipconfig_set_tcp_exponential_backoff_retry()

Prototype	A_INT32 custom_ipconfig_set_tcp_exponential_backoff_retry(	
	A_VOID handle,*	Driver context
	*A_UINT8 *v6addr,*	Pointer to ipv6 prefix
	A_INT32 retry,	Number of retries
	)	

2.2.21 custom_ipconfig_set_ip6_status()

Prototype	A_INT32 custom_ipconfig_set_ip6_status(	
	A_VOID handle,*	Driver context
	A_UINT16 enable,	0 or 1. IPv6 is enabled (1) by default.
	)	

Description	Enable/disable IPv6 module at run time. When IPv6 is disabled, the device does not transmit IPv6 packets including those related with Neighbor Discovery and drop any incoming IPv6 packets.
Return Value	0 on success, -1 on failure.

2.2.22 custom_ipconfig_dhcp_release

Prototype	A_INT32 custom_ipconfig_dhcp_release(	
	`A_VOID* handle,`	Driver context
	`)`	
Description	Release DHCP address.	
Return Value	0 on success, -1 on failure.	

2.2.23 custom_ipconfig_set_tcp_rx_buffer

Prototype	A_INT32 custom_ipconfig_set_tcp_rx_buffer(	
	`A_VOID* handle,`	Driver context
	`A_INT32 rxbuf )`	Number of Rx buffers
Description	Configure the number of Rx buffers to receive packets for a downlink TCP connection. TCP advertises zero Window when the application is to consume the configured number of Rx buffers.	
Return Value	0 on success, -1 on failure.	

2.2.24 custom_ip_http_server

Prototype	A_INT32 custom_ip_http_server(	
	`A_VOID* handle,`	Driver context
	`A_INT32 command`	1 (Start) or 0 (Stop)
	`)`	
Description	Start/stop the HTTP server.	
Return Value	0 on success, -1 on failure.	

2.2.25 custom_ip_httpc_method

Prototype	A_INT32 custom_ip_httpc_method(	
	A_VOID handle,*	Driver context
	A_INT32 command,	Connect/disc/get/post
	*A_UINT8 *url*	Page name
	*A_UINT8 *data*	Object name
	*A_UINT8 **output,*	Output of the HTTP Client Request
	)	
Description	Configure HTTP Client.	
Return Value	0 on success, -1 on failure.	

2.2.26 custom_ip_http_server_method

Prototype	A_INT32 custom_ip_http_server_method(	
	A_VOID handle,*	Driver context
	A_INT32 command,	Start/Stop
	*A_UINT8 *pagename,*	Page name from/to which the data is to get/post
	*A_UINT8 *objname,*	Objname name in HTML files
	A_INT32 objtype,	Type of the object (string/integer/Boolean)
	A_INT32 objlen,	Object length 1 (Boolean); 4 (integer); String length (string)
	*A_UINT8 * value*	Value of the object
	)	
Description	Get/post the values from/to the HTTP server. For GET operation, make sure sufficient memory is allocated before calling this API.	
Return Value	0 on success, -1 on failure.	

2.2.27 custom_ip_dns_client

Prototype	A_INT32 custom_ip_dns_client(	
	A_VOID handle,*	Driver context
	A_INT32 command	1 (Start), 0 (Stop)
	)	
Description	Start/stop DNS client.	
Return Value	0 on success, -1 on failure.	

2.2.28 custom_ip_resolve_hostname

Prototype	A_INT32 custom_ip_resolve_hostname (	
	A_VOID handle,*	Driver context
	*DNC_CFG_CMD *DncCfg*	This structure is filled by the application to resolve a domain name. The following modes can be selected: ▪ GETHOSTBYNAME ▪ GETHOSTBYNAME2 ▪ RESOLVEHOSTNAME. The domain can be either 1 (IPv4) and 2 (IPv6). The hostname specifies the name to resolve.
	*DNC_RESP_INFO *DncRespInfo*	Output is populated in the structure according to the specified mode.
	)	
Description	Resolve the hostname.	
Return Value	0 on success, -1 on failure.	

2.2.29 custom_ip_dns_server_addr

Prototype	A_INT32 custom_ip_dns_server_addr(	
	A_VOID handle,*	Driver context
	*IP46ADDR *addr*	IPv4 or IPv6 DNS server address to be configured
	)	
Description	Configure IPv4/IPv6 DNS server address.	
Return Value	0 on success, -1 on failure.	

2.2.30 custom_ip_dns_local_domain

Prototype	A_INT32 custom_ip_dns_local_domain(	
	A_VOID handle,*	Driver context
	*Char *domain_name*	Domain name to be configured
	)	
Description	Configure the domain name for the device.	
Return Value	0 on success, -1 on failure.	

2.2.31 custom_ip_hostname

Prototype	A_INT32 custom_ip_dns_local_domain(	
	A_VOID handle,*	Driver context
	*Char *domain_name*	Host name to be configured
	)	
Description	Configure the host name for the device.	
Return Value	0 on success, -1 on failure.	

2.2.32 custom_ip_dns

Prototype	A_INT32 custom_ip_dns(	
	A_VOID handle,*	Driver context

	*Char *domain_name*	Domain name to be populated in the data base
	*IP46ADDR *dnsaddr*	Pointer to IPv4/IPv6 address
	)	
Description	Configure DNS server database.	
Return Value	0 on success, -1 on failure.	

2.2.33 custom_ip_sntp_srvr_addr

rototype	A_INT32 custom_ip_sntp_srvr_addr(	
	*Void *handle*	Driver context
	*Char *sntp_srvr_addr*	Pointer to SNTP Server address
	)	
Description	Configure SNTP Server address.	
Return Value	0 on success, -1 on failure.	

2.2.34 custom_ip_sntp_get_time

Prototype	A_INT32 custom_ip_sntp_get_time	
	*Void *handle*	Driver Context
	*tSntpTime *SntpTime*	Pointer to SNTP time structure
	)	
Description	Get SNTP time.	
Return Value	0 on success, -1 on failure.	

2.2.35 custom_ip_sntp_get_time_of_day

Prototype	A_INT32 custom_ip_sntp_get_time_of_day(	
	*void *handle*	Driver context
	*tSntpTM *SntpTM*	Pointer to SNTP structure that contains time in seconds and milliseconds

	)	
Description	Get SNTP time of day in seconds and milliseconds.	
Return Value	0 on success, -1 on failure.	

2.2.36 custom_ip_sntp_modify_zone_dse

Prototype	A_INT32 custom_ip_sntp_modify_zone_dse(	
	*Void *handle*	Driver context
	A_UINT8 hour	Time stored in hours
	A_UINT8 min	Time stored in minutes
	A_UINT8 add_sub	Time to add or subtract to get the appropriate time zone
	A_UINT8 dse	Enable or disable day light saving
	)	
Description	Configure SNTP time zone and enable/disable day light saving.	
Return Value	0 on success, -1 on failure.	

2.2.37 custom_ip_sntp_query_srvr_address

Prototype	A_INT32 custom_ip_sntp_query_srvr_address(	
	*Void *handle*	Driver context
	tSntpDNSAddr SntpDnsAddr	Structure containing SNTP server addresses
	)	
Description	Query the SNTP address.	
Return Value	0 on success, -1 on failure.	

2.2.38 custom_ip_sntp_client

Prototype	A_INT32 custom_ip_sntp_client (	
	*void *handle*	Driver context
	A_INT32 command	Enable/disabe SNTP

	)	
Description	Enable/disable SNTP at run time.	
Return Value	0 on success, -1 on failure.	

2.2.39 SSL_ctx_new

Prototype	SSL_CTX *SSL_Ctx_new(	
	SSL_ROLE_T role	1 (server), 2 (client)
	A_INT32 inbufsize	Initial Inbuf size
	A_INT32 outbufsize	Outbuf size
	A_INT32 reserved	Not used, must be 0
	)	
Description	Creates SSL context. This function must be called (as either server or client) before using any other SSL functions.	
Return Value	SSL context handle on success, NULL on error (out of memory)	

2.2.40 SSL_ctx_free

Prototype	A_INT32 SSL_ctx_free(	
	*SSL_CTX *ctx*	SSL context
	)	
Description	Free SSL context.	
Return value	0 on success, -1 on failure.	

2.2.41 SSL_new

Prototype	SSl *SSL_new(	
	*SSL_CTX *ctx*	SSL context
	)	
Description	Create SSL connection object.	
Return value	SSL object handle on success, NULL on error (out of memory)	

2.2.42 SSL_set_fd

Prototype	A_INT32 SSL_set_fd(	
	*SSL *ssl*	SSL Connection
	A_UINT32 fd	Socket descriptor
	)	
Description	Attach TCP socket descriptor to SSL connection.	
Return Value	1 on success, negative error code on errors	

2.2.43 SSL_accept

Prototype	A_INT32 SSL_accept(	
	*SSL *ssl*	SSL connection
	)	
Description	Initiate SSL handshake (when acting as server). This API is invoked by SSL Server to respond to the incoming SSL Handshake from the client.	
Return Value	1 on success, ESSL_HSDONE if handshake has already been performed, negative error code otherwise.	

2.2.44 SSL_connect

Prototype	A_INT32 SSL_connect(	
	*SSL *ssl*	SSL connection
	)	
Description	Initiate SSL handshake (when acting as client).	
Return Value	1 on success, ESSL_HSDONE if handshake has already been performed, negative error code otherwise.	

2.2.45 SSL_shutdown

Prototype	A_INT32 SSL_shutdown(	
	*SSL *ssl*	SSL connection
	)	
Description	Close the SSL connection (when acting as client).	

Return Value	1 on success, negative error code on error

2.2.46 SL_configure

Prototype	A_INT32 SSL_configure(	
	*SSL *ssl*	SSL connection
	*SSL_CONFIG *cfg)*	Pointer to the structure that contains the SSL configuration
	)	
Description	Configure SSL connection.	
Return Value	1 on success, negative error code on error	

2.2.47 SSL_setCaList

Prototype	A_INT32 SSL_setCaList(	
	*SSL_CTX *ctx,*	SSL Context
	sslCAList calist	Reference to SSL CA LIST
	A_UINT32 size	Size of the array
	)	
Description	Set a CA list. SSL performs certificate validation based on the peer certificate. Only one CA list is allowed, thus the CA list must include all root certificates required for the session.	
Return Value	1 on success, negative error code on error	

2.2.48 SSL_addCert

Prototype	A_INT32 SSL_addCert(	
	*SSL_CTX *ctx*	Pointer to SSL context
	Sslcert cert	Address of array of binary data
	A_UINT32 size	Size of array
	)	

Description	Add a certificate to the SSL object. SSL object in server mode requires at least one certificate.
Return Value	1 on success, negative error code on error

2.2.49 SSL_storeCert

Prototype	A_INT32 SSL_storeCert(	
	*A_CHAR *name*	Name of the certificate
	Sslcert cert	Address of array of binary data
	A_UINT32 size)	Size of array
	)	
Description	Store a certificate or CA list to flash.	
Return Value	1 on success, negative error code on error	

2.2.50 SSL_loadCert

Prototype	A_INT32 SSL_loadCert(	
	*SSL_CTX *ctx*	Pointer to SSL Context
	SSL_CERT_TYPE_T type	type - Certificate or CA List
	*Char *name)*	Name of the certificate
	)	
Description	Load a certificate or CA list from flash and store it in the SSL object.	
Return Value	1 on success, negative error code on error	

2.2.51 SSL_read

Prototype	A_INT32 SSL_read(	
	*SSL *ssl*	Pointer to SSL connection object
	*Void **buf*	Pointer to the pointer holding the address of the receive buffer

	A_INT32 num	The maximum number of bytes to read
	)	
Description	Read data over SSL connection. This is the ZERO_COPY version of SSL read function. It uses the SSL pointer to find the socket to read, and check the socket receive queues for pending packets. When a packet is available, it is passed to the application without a memcopy. The application must call zero_copy_free to free this buffer.	
Return Value	Number of bytes received on success, A_ERROR on failure	

2.2.52 SSL_write

Prototype	A_INT32 SSL_write(	
	*SSL *ssl*	Pointer to SSL connection object
	*Const void *buf*	pointer to buffer holding the data to send
	A_INT32 num	The number of bytes to send
	)	
Description	Write data over SSL connection. This function encrypts and sends the data in buffer on the associated socket. The SSL pointer is used to find the socket.	
Return Value	Number of bytes sent on success, negative error code on error	

API usage

This section provides details on the usage of various APIs to send and receive UDP and TCP traffic. It also illustrates examples of using the API.

UDP transmit

1. Open socket.

Create an IPv4 UDP socket:

```
socket_foreign = t_socket((void*)handle, ATH_AF_INET, SOCK_DGRAM_TYPE, 0)
```

Create an IPv6 socket.

```
socket_foreign = t_socket((void*)handle, ATH_AF_INET6, SOCK_DGRAM_TYPE, 0)
```

The same API is used in both cases. Only the second parameter is different.

2．Connect to peer. The *connect* API can be used after a valid socket handle is acquired (not mandatory for UDP transmit):

```
t_connect ((void*)handle, socket_foreign, (void*)(&foreign_addr),
sizeof(foreign_addr))
```

Where *foreign_addr* can be of type SOCKADDR_T (IPv4) or SOCKADDR_6_T (IPv6) and must be populated with the port and IP address of the peer.

3.Obtain buffer for transmission.

```
while((buffer = CUSTOM_ALLOC(size)) == NULL)
{
/*Wait till we get a buffer*/
/*If necessary, allow small delay to allow other thread to run*/
 _time_delay(SMALL_TX_DELAY);
}
```

4.Send datagrams.

```
result = t_sendto((void*)handle, socket_foreign, (unsigned
 char*)buffer, size, 0,
(void*)(&foreign_addr), sizeof(foreign_addr))
```

5.Close socket.

Once all packets are transmitted, this API must be used to close the socket. Due to the limited number of sockets, it is imperative to close the sockets after use.

```
t_shutdown ((void*)handle, socket_foreign)
```

6.Open socket.

Create an IPv4 UDP socket:

```
socket_foreign = t_socket((void*)handle, ATH_AF_INET, SOCK_DGRAM_TYPE, 0)
```

Create an IPv6 socket.

```
socket_foreign = t_socket((void*)handle, ATH_AF_INET6, SOCK_DGRAM_TYPE, 0)
```

The same API is used in both cases. Only the second parameter is different.

7.Connect to peer. The *connect* API can be used after a valid socket handle is acquired (not mandatory for UDP transmit):

```
t_connect ((void*)handle, socket_foreign, (void*)(&foreign_addr),
sizeof(foreign_addr))
```

Where *foreign_addr* can be of type SOCKADDR_T (IPv4) or SOCKADDR_6_T (IPv6) and must be populated with the port and IP address of the peer.

8.Obtain buffer for transmission.

```
while((buffer = CUSTOM_ALLOC(size)) == NULL)
{
/*Wait till we get a buffer*/
/*If necessary, allow small delay to allow other thread to run*/
 _time_delay(SMALL_TX_DELAY);
}
```

9.Send datagrams.

```
result = t_sendto((void*)handle, socket_foreign, (unsigned
                char*)buffer, size, 0,
(void*)(&foreign_addr), sizeof(foreign_addr))
```

10.Close socket.

Once all packets are transmitted, this API must be used to close the socket. Due to the limited number of sockets, it is imperative to close the sockets after use.

```
t_shutdown ((void*)handle, socket_foreign)
```

UDP receive

1.Open socket.

Create an IPv4 UDP socket:

```
socket_foreign = t_socket((void*)handle, ATH_AF_INET, SOCK_DGRAM_TYPE, 0)
```

Create an IPv6 socket:

```
socket_foreign = t_socket((void*)handle, ATH_AF_INET6, SOCK_DGRAM_TYPE, 0)
```

The same API is used in both cases. Only the second parameter is different.

2.Bind socket.

To receive incoming datagrams, the socket must bind to a local interface. This interface is identified by *local_addr* variable, which can be of type SOCKADDR_T (IPv4) or SOCKADDR_6_T (IPv6).

```
t_bind((void*)handle, socket_listen, (void*)(&local_addr),
sizeof(local_addr))
```

3.Wait for incoming traffic (select).

A *recvfrom* call is blocking, so *select* can be used to periodically check for incoming packets. This call is implemented in the driver and is not conveyed to the firmware. Select can be called for a fixed time period (specified in milliseconds). It returns A_OK if activity is detected or A_ERROR otherwise. If a packet arrives, the select call returns immediately instead of waiting for entire duration.

```
result = t_select ((void*)handle, socket_listen, timeout)
```

4.Receive packet.

If select returns success, the following API can be used to retrieve the queued packet. The field a*ddr* is used to distinguish IPv4 from IPv6 with the type SOCKADDR_T (IPv4) or SOCKADDR_6_T (IPv6). The API returns number of packets.

```
received = t_recvfrom((void*)handle, socket_listen, buffer,
BUFFER_SIZE, 0,&addr,
(A_UINT32*)&addr_len);
```

5.Close socket.

Once all packets are transmitted, this API must be used to close the socket. Due to the limited number of sockets, it is imperative to close the sockets after use.

```
t_shutdown ((void*)handle, socket_foreign)
```

TCP transmit

1.Open socket.

Create an IPv4 TCP socket:

```
socket_foreign = t_socket((void*)handle, ATH_AF_INET, SOCK_STREAM_TYPE, 0)
```

Create an IPv6 socket:

```
socket_foreign = t_socket((void*)handle, ATH_AF_INET6, SOCK_STREAM_TYPE, 0)
```

2.Connect to peer. The *connect* API can be used after a valid socket handle is acquired (not mandatory for TCP transmit):

```
t_connect ((void*)handle, socket_foreign, (void*)(&foreign_addr),
sizeof(foreign_addr))
```

Where *foreign_addr* can be of type SOCKADDR_T (IPv4) or SOCKADDR_6_T (IPv6) and must be populated with the port and IP address of the peer.

3.Obtain buffer for transmission.

```
while((buffer = CUSTOM_ALLOC(size)) == NULL)
{
 /*Wait till we get a buffer*/
/*If necessary, allow small delay to allow other thread to run*/
_time_delay(SMALL_TX_DELAY);
}
```

4.Send packets.

```
result = t_send((void*)handle, socket_foreign,
(unsigned char*)buffer, size, 0)
```

5.Close socket.

Once all packets are transmitted, this API must be used to close the socket. Due to the limited number of sockets, it is imperative to close the sockets after use.

```
t_shutdown ((void*)handle, socket_foreign)
```

TCP receive

1.Open socket.

Create an IPv4 TCP socket:

```
socket_foreign = t_socket((void*)handle, ATH_AF_INET,
SOCK_STREAM_TYPE, 0)
```

Create an IPv6 socket:

```
socket_foreign = t_socket((void*)handle, ATH_AF_INET6,

SOCK_STREAM_TYPE, 0)
```

The same API is used in both cases. Only the second parameter is different.

2.Listen.

Once socket is created, it must be put into listening state.

```
t_listen ((void*)handle, socket_listen, 1)
```

3.Wait for incoming connection.

The *accept* blocks the application thread until a connection is available. To avoid this, the *select* call can be used to check for incoming connections periodically.

```
result = t_select ((void*)handle, socket_listen, timeout)
```

4.Accept incoming connection.

If select returns success, call *accept* to receive incoming connection. Accept returns the new socket handle which will be used to receive packets. The field *foreign_addr* is populated with the IP information of the peer.

```
socket_foreign = t_accept((void*)handle, socket_listen,
&foreign_addr, addr_len)
```

5.Wait for incoming packets.

Because *recv* call is blocking, *select* can be used in conjunction with *recv* to detect and receive incoming packets.

```
while(1)
{
 result = t_select((void*)handle, socket_foreign, timeout);
if(result == A_OK)
{         //Packet is available, receive it
received = t_recv((void*)handle, socket_foreign, buffer,
                BUF_SIZE, 0);
if(received == A_SOCK_INVALID)
{ //Test ended, peer closed connection
 Break;
}
 else
  {                 //Process packet
 }
 }
}
```

6.Close socket.

Once all packets are transmitted, this API must be used to close the socket. Due to the limited number of sockets, it is imperative to close the sockets after use..

```
t_shutdown ((void*)handle, socket_foreign)
```

SSL transmit

Open socket

1.Create an IPv4 TCP socket:

```
socket_foreign = t_socket((void*)handle, ATH_AF_INET, SOCK_STREAM_TYPE, 0)
```

Create an IPv6 socket:

```
socket_foreign = t_socket((void*)handle, ATH_AF_INET6, SOCK_STREAM_TYPE, 0)
```

2.Connect to peer.

The *connect* API can be used after a valid socket handle is acquired (not mandatory for TCP transmit):

```
t_connect ((void*)handle, socket_foreign,
(void*)(&foreign_addr),  sizeof(foreign_addr))
ssl->ssl = SSL_new(ssl->sslCtx);
result = SSL_set_fd(ssl->ssl, sock_foreign);
    if (result < 0)
{
 printf("ERROR: Unable to add socket handle"
 "to SSL (%d)\n", result);
 goto ERROR_2;
     }
// SSL handshake with server
 result = SSL_connect(ssl->ssl);
```

Where *foreign_addr* can be of type SOCKADDR_T (IPv4) or SOCKADDR_6_T (IPv6) and must be populated with the port and IP address of the peer.

3．Obtain buffer for transmission.

```
while((buffer = CUSTOM_ALLOC(size)) == NULL)
{
/*Wait till we get a buffer*/
/*If necessary, allow small delay to allow other thread to run*/
  _time_delay(SMALL_TX_DELAY);}
```

4.Send packets.

```
send_result = SSL_write(ssl->ssl, buffer, packet_size);
```

5.Close socket.

Once all packets are transmitted, this API must be used to close the socket. Due to the limited number of sockets, it is imperative to close the sockets after use.

```
SSL_shutdown(ssl->ssl);

t_shutdown ((void*)handle, socket_foreign)
```

SSL receive

1.Open socket.

Create an IPv4 TCP socket:

```
socket_foreign = t_socket((void*)handle, ATH_AF_INET, SOCK_STREAM_TYPE, 0)
```

Create an IPv6 socket:

```
socket_foreign = t_socket((void*)handle, ATH_AF_INET6, SOCK_STREAM_TYPE, 0)
```

The same API is used in both cases. Only the second parameter is different.

2.Listen.

Once socket is created, it must be put into listening state.

```
t_listen ((void*)handle, socket_listen, 1)
```

3.Wait for incoming connection.

Accept call will block the application thread until a connection is available. To avoid this, select can be used to check for incoming connections. As mentioned previously, select can be called periodically to check for activity.

```
result = t_select ((void*)handle, socket_listen, timeout)
```

4.Accept incoming connection.

If select returns success, call *accept* to receive incoming connection. Accept returns the new socket handle which will be used to receive packets. The field *foreign_addr* is populated with the IP information of the peer.

```
socket_foreign = t_accept((void*)handle, socket_listen, &foreign_addr,
addr_len)
// Create SSL connection object
                ssl->ssl = SSL_new(ssl->sslCtx);
// Add socket handle to SSL connection
                result = SSL_set_fd(ssl->ssl, socket_foreign);
// SSL handshake with server
                result = SSL_accept(ssl->ssl);
```

5.Wait for incoming packets.

Because *recv* call is blocking, *select* can be used in conjunction with *recv* to detect and receive incoming packets.

```
while(1)

{
   result = t_select((void*)handle, socket_foreign, timeout);
  if(result == A_OK)
  {        //Packet is available, receive it
      received = SSL_read(ssl->ssl,
                  (void**)& buffer,BUF_SIZE);
    if(received == A_SOCK_INVALID)
   { //Test ended, peer closed connection
    break;
   }
     else
     {                //Process packet
     }
  }
}
```

6.Close socket.

Once all packets are transmitted, this API should be used to close the socket. Note that due to limited number of sockets, it is imperative to close sockets after use.

```
SSL_shutdown(ssl->ssl);
t_shutdown ((void*)handle, socket_foreign)
```

Multicast support

Multicast is currently supported for IPv4 and IPv6. These multicast options are supported:

```
/*Multicast options*/
#define  IP_ADD_MEMBERSHIP   12 /* ip_mreq; add an IP group membership */
#define  IP_DROP_MEMBERSHIP 13 /* ip_mreq; drop an IP group membership */
/*IPv6*/
#define IPV6_JOIN_GROUP            83 /* ipv6_mreq; join MC group */
#define IPV6_LEAVE_GROUP        84 /* ipv6_mreq; leave MC group */
```

To add or drop multicast group membership, the following structure is passed to the target firmware.

```
/* IP v4*/
typedef struct _ip_mreq
{
   A_UINT32 imr_multiaddr;    /*Multicast group address*/
  A_UINT32 imr_interface;      /*Interface address*/
} IP_MREQ_T;
/*IP v6*/
  typedef  struct ipv6_mreq {
  IP6_ADDR_T ipv6mr_multiaddr;    /* IPv6 multicast address */
  IP6_ADDR_T ipv6mr_interface;    /* IPv6 interface address */
} IPV6_MREQ_T;
```

Sample usage for IPv4:

```
IP_MREQ_T mreq;
mreq.imr_multiaddr = A_CPU2BE32(multicast_addr);
mreq.imr_interface = A_CPU2BE32(interface_addr);
if(t_setsockopt((void*)handle, sockHandle, ATH_IPPROTO_IP,
IP_ADD_MEMBERSHIP,
  (A_UINT8*)(&mreq),sizeof(IP_MREQ_T)) != A_OK)
 {
   printf("SetsockOPT error : unable to add to multicast group\r\n");
    return A_ERROR;
}
```

Sample usage for IPv6:

```
IPV6_MREQ_T mreq;
A_MEMCPY(mreq.ipv6mr_multiaddr, multicast_addr, sizeof(IP6_ADDR_T));
A_MEMCPY(mreq.ipv6mr_interface , interface_addr, sizeof(IP6_ADDR_T));
if(t_setsockopt((void*)handle, sockHandle, ATH_IPPROTO_IP,
              IPV6_JOIN_GROUP,
             (A_UINT8*)(&mreq), sizeof(IPV6_MREQ_T)) != A_OK)
{
    printf("SetsockOPT error : unable to add to multicast group\r\n");
  return A_ERROR;
}
```

NOTE: Both multicast address and interface address must be in network order. Using platform independent *A_CPU2BE32* macro can ensure this.

Supported Socket Options

Socket options can be set or retrieved using *t_setsockopt* and *t_getsockopt* APIs. These socket options are supported:

```
)
TCP_MAXSEG       //Set maximum segment size
IP_ADD_MEMBERSHIP //Takes ip_mreq as parameter; add IP group membership
IP_DROP_MEMBERSHIP //Takes ip_mreq as parameter; drop IP group membership
```

Miscellaneous

Host driver error

A global variable in the driver last_driver_error stores the last error encountered in the driver. It can be used for debugging purposes. The definitions for different driver errors are available in athdefs.h.

Optimizing memory usage

To reduce driver RAM usage, following steps can be taken-

- Reduce the number of scan buffers (defined in atheros_wifi_api.h)
- If store-recall feature is not needed, compile it out (defined in a_config.h).
- Reduce the number of sockets, depending upon the application (defined in atheros_stack_offload.h)
- Reduce the number of buffers in buffer pool. Multiple buffers of varying sizes form a buffer pool from which buffers are obtained by the driver and application using malloc. The number and size of buffers is currently selected to allow two simultaneous uplink streams of traffic. This number and size combination can be adjusted depending upon application requirements.

NOTE: Users must be extremely careful in changing these numbers as it may lead to a memory allocation failure in the driver. This pool is defined in **cust_driver_netbuf.c**.

Summary of design considerations

Application developers/designers should keep these points in mind while writing an application:

- All transmit buffers/packets must be obtained via a call to CUSTOM_ALLOC and freed by calling CUSTOM_FREE.
- If ZERO_COPY option is enabled, received packet must be freed by calling *zero_copy_free*.
- Do not try to transmit a received buffer directly as it may lead to deadlocks. Always copy it into a TX buffer. A sample UDP echo server application is provided in *throughput_demo* example project for reference.
- Always call *t_select* to check for incoming data or TCP connections. If *t_select* is

successful, call *t_recv/t_recvfrom* or *t_accept* respectively. This will prevent the application thread from being blocked. For reference implementation, look at *ath_udp_recv* and *ath_tcp_recv* APIs in *throughput.c* file in *throughput_demo* sample application.

3 Host MCU IP Stack

This chapter describes the host driver changes required to support a Host MCU-based IP stack. The host driver exposes a set of APIs to Host MCU IP stack that can be used to send and receive Ethernet packets. These APIs must be customized according to the target platform and the operating system. The macro ENABLE_STACK_OFFLOAD in a_config.h must be disabled when compiling the driver sources.

The APIs and their functionality are summarized in Table 3-1.

Table 3-1 Host MCU IP stack APIs

API	Description
3.1.1 Custom_Api_Initialize	Entry point for initializing the driver
3.1.2 Custom_Api_Shutdown	Entry point for to shutdown the driver

API	Description
3.1.3 Custom_Api_Send	Entry point for driver interface to send a packet
3.1.4 Custom_Api_Mediactl	Entry point for configuring the WiFi interface
3.1.5 Custom_DeliverFrameToNetworkStack	Driver function which calls the IP stack receiver function to send the received Ethernet packets to the IP stack

3.1.1 Custom_Api_Initialize

Definition	Entry point for initializing the driver	
Prototype	`A_INT32 Custom_Api_Initialize (`	
	`A_VOID** handle`	Driver context
	`}`	
Description	This does the following: ▪ Allocates memory for the driver context. ▪ Creates events/mutexes required for synchronization between the IP stack, driver task and the SPI driver interfacing with the Wi-Fi chip. ▪ Initializes the necessary data structures. ▪ Starts the driver task, waits for the driver to initialize the Wi-Fi chip. ▪ Returns error code, success/failure.	

3.1.2 Custom_Api_Shutdown

Definition	Entry point for to shutdown the driver	
Prototype	`A_INT32 Custom_Api_Shutdown (`	
	`A_VOID* pCxt`	Driver context
	`}`	
Description	This does the following ▪ Shuts down the Wi-Fi chip ▪ Destroy the events/mutexs ▪ Free up memory used by the driver	

3.1.3 Custom_Api_Send

Definition	Entry point for driver interface to send a packet	
Prototype	`A_INT32 Custom_Api_Send (`	
	*A_VOID** handle,*	Driver context
	A_UINT8 pBuffer,*	Pointer to the buffer containing payload data
	A_UINT32 size	Size of the payload
	`}`	
Description	This function converts the packet into a format the generic driver can understand and calls the Api_DataTxStart() function. The Api_DataTxStart() function sends the packet to the WiFi chip. Note that the function Api_DataTxStart() will block till the packet is sent down to the WiFi chip.	

3.1.4 Custom_Api_Mediactl

Definition	Entry point for configuring the WiFi interface	
Prototype	`A_INT32 Custom_Api_MediaCtl (`	
	`ENET_CONTEXT_STRUCT_PTR enet_ptr,`	MQX Enet structure pointer
	A_INT32 commandId,	IOCTL command ID
	A_VOID inout_param*	Parameter associated with this command
	`}`	
Description	This function converts the OS and Platform specific media control operation to QCA specific WMI messages. Note that standard Media control operations defined by the target OS may not directly map to WMI messages. In such cases, it is recommended to use vendor specific Media Control IOCTLs.	

3.1.5 Custom_DeliverFrameToNetworkStack

Definition	Driver function which calls the IP stack receiver function to send the received Ethernet packets to the IP stack**.**	
Prototype	`A_VOID Custom_DeliverFrameToNetworkStack (`	
	`A_VOID** pCxt`	Driver context
	`A_VOID** pReq`	Pointer to the MAC frame to be delivered to the network stack
	`}`	
Description	This function is called by the driver upon receiving a pcaket from the WiFi chip. This function calls the IP stack function for handlig the received packet. This function calls the Host MCU receiver function for handling the received MAC frame.	

4 Wireless Module Interface

This section summarizes the format and usage model for WMI control and data messages between the host and the target. The header file **include/wmi.h** contains all command and event codes, constants, as well as structure typedefs for each set of command and reply parameters.

4.1 Data frames

The data payload transmitted and received by the target follows RFC-1042 encapsulation and thus starts with an 802.2-style LLC-SNAP header. The WLAN module completes 802.11 encapsulation of the payload, including the MAC header, FCS, and WLAN security related fields. At the interface to the message transport (HTC), a data frame is encapsulated in a WMI message.

4.2 WMI message structure

WMI protocol leverages an 802.3-style Ethernet header in communicating the source and destination information between the host and the target modules using a 14-byte 802.3 header ahead of the 802.2-style payload. In addition, WMI protocol adds a header to all data messages:

```
{
A_INT8         rssi;
The RSSI of the received packet and its units are shown in dB above the noise
floor, and the noise floor is shown in dBm.


A_UINT8   info;
Contains information on message type and user priority.
Message type differentiates between a data packet and a synchronization
message.
b1:b0      - WMI_MSG_TYPE
0: Data; 1: Control; 2: SYNC; 3: OPT
b4:b3:b2    - UP(tid)
b5         - AP mode: More-data in Tx direction, PS in Rx.
b7:b6      - 0: Dot3 hdr; 1:Dot11 hdr; 2: ACL data

A_UINT16  info2;
Contains information on sequence number used and other optional parameters
b11:b0     - seq_no
b12        - A-MSDU?
b15:b13     - META_DATA_VERSION 0 - 7

A_UINT16  info3;
Contains the device ID
b3:b2:b1:b0 - device id
       } WMI_DATA_HDR
```

User priority contains the 802.1d user priority info from host to target. Host software translates the host Ethernet format to 802.3 format prior to Tx and 802.3 format to host format in the Rx

direction. The host does not transmit the FCS that follows the data. `msgType` differentiates between a regular data packet (`msgType`=0) and a synchronization message (`msgType`=1).

NOTE: The QCA4002 host driver uses a META_DATA_VERSION of 3 (WMI_META_VERSION_3) to support the raw Tx mode used to send frames with arbitrary 802.11 MAC configured by an application running on the host. The format of the META data is defined in wmi.h.

NOTE: // WMI_TX_RATE_SCHEDULE specifies a user-provided rate schedule to replace
// what would be normally determined by firmware.   This allows the host to
// specify what rates and attempts should be used to transmit the frame

```
NOTE:  #define WMI_TX_MAX_RATE_SERIES (4)
NOTE:  typedef struct {
NOTE:          // rate index for each series. first invalid rate terminates series.
               // where 0==1mbps; 1==2mbps; 2==5.5mbps; …
               A_UINT8 rateSeries[WMI_TX_MAX_RATE_SERIES];
NOTE:          //number of tries for each series (1 – 14)
               A_UINT8 trySeries[WMI_TX_MAX_RATE_SERIES];
NOTE:          // combination of WMI_META_TX_FLAG...
               A_UINT8 flags FIELD_PACKED;
NOTE:          // should be WMM_AC_BE for management frames and multicast frames.
               A_UINT8 accessCategory FIELD_PACKED;
NOTE:  } WMI_TX_RATE_SCHEDULE;
NOTE:
NOTE:  typedef struct {
NOTE:          WMI_TX_RATE_SCHEDULE rateSched;
NOTE:          // The packet ID to identify the Tx request
               // 0=beacon frame; 1=QOS data frame; 2=4 address data frame
               A_UINT8         pktID;
NOTE:    } WMI_TX_META_V3;
```

NOTE:The QCA4002 host driver uses a META_DATA_VERSION of 5 (WMI_META_VERSION_5) to support socket downlink data while using the IP Offload stack on the QCA4002. This allows for the efficient dispatching of the data to the IP offload module in the firmware. A socket header is added for data frames in the uplink direction, or added by the IP Offload stack to the host in the download direction that provides information between the driver and the target that help it determine which socket the packet is bound to or coming from.

```
// This structure is sent to the target in a data packet.
// It allows the target to route the data to correct socket with
// all the necessary parameters*/
typedef struct sock_send
{
   A_UINT32 handle;              //Socket handle
   A_UINT16 length;              //Payload length
   A_UINT16 reserved;       //Reserved
   A_UINT32 flags;          //Send flags
   SOCKADDR_T name;         //IPv4 destination socket information
   A_UINT16 socklength;
} SOCK_SEND_T;

typedef struct sock_send6
{
   A_UINT32 handle;              //Socket handle
   A_UINT16 length;              //Payload length
   A_UINT16 reserved;            //Reserved
   A_UINT32 flags;          //Send flags
   SOCKADDR_6_T name6;      //IPv6 destination socket information
   A_UINT16 socklength;
} SOCK_SEND6_T;
```

```
typedef struct sock_recv
{
   A_UINT32 handle;               //Socket handle
   SOCKADDR_T name;          //IPv4 destination socket information
   A_UINT16 socklength      // Length of sockaddr structure
   A_UINT32 reassembly_info;      //Placeholder for reassembly info
} SOCK_RECV_T;

typedef struct sock_recv6
{
   A_UINT32 handle;               //Socket handle
   SOCKADDR_6_T name6;            //IPv6 destination socket information
   A_UINT16 socklength;
   A_UINT16 reserved;        //Reserved
   A_UINT32 reassembly_info;      //Placeholder for reassembly info
} SOCK_RECV6_T;

typedef struct _ip_mreq
{
  A_UINT32 imr_multiaddr;         //Multicast group address
  A_UINT32 imr_interface;         //Interface address
} IP_MREQ_T;

typedef struct ipv6_mreq {
  IP6_ADDR_T ipv6mr_multiaddr;    // IPv6 multicast addr
  IP6_ADDR_T ipv6mr_interface;    // IPv6 interface address
} IPV6_MREQ_T;
```

4.3 Data endpoints

The target chipset provides several data endpoints to support quality of service (QoS) and maintains separate queues and separate DMA engines for each data endpoint. Data endpoints are abstract message pipes provided by the HTC layer. The HTC layer is responsible for transferring messages to the chipset. A data endpoint can be bi-directional.

NOTE: The QCA4002   host driver does not enable full support for QoS and uses Endpoint 2 for all data. This is to ensure minimizing the driver space memory requirements for a restricted host platform.

4.4 Connection states

Table 4-1 describes the firmware WLAN connection states:

Table 4-1   Connection states

Connection State	Description

Connection State	Description
DISCONNECTED	In this state, the QCA4002 device is not connected to a wireless network. The device is in this state after reset when it sends the **WMI “READY” EVENT**, after it processes a **DISCONNECT** command, and when it loses its link with the access point (AP) that it was connected to. The device signals a transition to the DISCONNECTED state with a **“DISCONNECT”** event.
CONNECTED	In this state, the QCA4002 device is connected to wireless networks. The device enters this state after successfully processing a **CONNECT**, which establishes a connection with a wireless network. The device signals a transition to the CONNECTED state with a **“CONNECT”** event.

4.5 Message types

WMI uses commands, replies, and events for the control and configuration of the target device. The control protocol is asynchronous. Table 4-2 describes target message types:

Table 4-2 Message types

Message Type	Description
Commands	Control messages that flow from the host to the device
Replies/Events	Control messages that flow from the device to the host. The device issues a reply to some WMI commands, but not to others. The payload in a reply is command-specific, and some commands do not trigger a reply message at all. Events are control messages issued by the device to signal the occurrence of an asynchronous event.

4.6 WMI message format

All WMI control commands, replies, and events use the header format:

```
{
A_UINT16  id;  This 16-bit constant identifies which WMI command the host is
issuing, which command the target is replying to, or which
 event has occurred.
WMI_CMD_HDR
} WMI_CMD_HDR
```

A variable-size command-, reply-, or event-specific payload follows the header. Over the interconnect, all fields in control messages (including WMI_CMD_HDR and the command specific payload) use 32-bit little Endian byte ordering and fields are packed. The QCA4002 device always executes commands in order, and the host may send multiple commands without waiting for previous commands to complete. A majority of commands are processed to completion once received. Other commands trigger a longer duration activity whose completion is signaled to the host through an event.

4.7 Command restrictions

Some commands may only be issued when the QCA4002 device is in a certain state. The host is required to wait for an event signaling a state transition before such a command can be issued. For example, if a command requires the device to be in the CONNECTED state, then the host is required to wait for a "CONNECT" event before it issues that command.

The device ignores any commands inappropriate for its current state. If the command triggers a reply, the device generates an error reply. Otherwise, the device silently ignores the inappropriate command.

4.8 WMI commands

Command Name	Value	Description
4.8.1 ADD_CIPHER_KEY	0x0016	Add or replace any of the four target's encryption keys
4.8.2 ALLOW_AGGR	0xF01B	Configure TIDs for which aggregation is allowed
4.8.51 AP_CONFIG_COMMIT	0xF00F	Set QCA4002 to Soft AP mode
4.8.52 AP_HIDDEN_SSID	0xF00B	Set SSID to Hidden Mode in Soft AP functionality
4.8.53 AP_INACTIVITY_TIME		Set the inactivity timeout value in Soft AP functionality
4.8.56 AP_PSBUFF_OFFLOAD	0xF0AF	Enable power save buffering and set number of buffers when handling power save clients in Soft AP mode
4.8.55 AP_SET_COUNTRY_CODE	0xF014	Set the Country Code in SoftAP mode
4.8.54 AP_SET_DTIM_INTERVAL	0xF015	Set DTIM interval in SoftAP mode

Command Name	Value	Description
4.8.3 CONNECT	0x0001	Request the target device establish wireless connection with a SSID
4.8.4 DELETE_CIPHER_KEY	0x0017	Delete a previously added cipher key
4.8.5 DISCONNECT	0x0003	Disconnect from the associated AP or abort connection process
4.8.50 GET_BITRATE	0xF001	Get Bit Rate of the last transmitted frame
4.8.6 GET_CHANNEL_LIST	0x00E	Retrieve the list of channels used by the target
4.8.7 GET_KEEPALIVE	0x003E	Used to get the configured keepalive interval
4.8.8 GET_PMK	0xf047	Retrieve the PMK from the firmware
4.8.9 GET_TX_PWR	0x001C	Retrieves the current transmit power
4.8.10 GET_WPS_STATUS	0xF054	Returns the current status of WPS
4.8.11 P2P_AUTH_GO_NEG	0xF093	Authorize P2P Device in the P2P Node list
4.8.12 P2P_CANCEL	0xF05D	Cancel current P2P Session
4.8.13 P2P_CONNECT	0xF091	Connect to P2P Device
4.8.14 P2P_FIND	0xF03B	Start P2P Device Discovery
4.8.15 P2P_FW_PROV_DISC_REQ	0xF0AA	Send P2P Provisional request to a P2P Device
4.8.16 P2P_GET_NODE_LIST	0xF092	Request firmware to send P2P device list populated during discovery
4.8.17 P2P_GRP_INIT	0xF051	Indicate firmware to initialize settings for P2P GO
4.8.18 P2P_INVITE_REQ_RSP	0xF054	P2P GO Inviting P2P Device to join P2P network
4.8.24 P2P_LIST_PERSISTENT_NETWORK		Retrieve stored P2P persistent network connections
4.8.19 P2P_SET	0xF056	Set SSID Post-Fix , MODE, and GO intent
4.8.20 P2P_SET_CONFIG	0xF038	Set P2P Configuration , Intent value, country code, operating channel etc.
4.8.21 P2P_SET_JOIN_PROFILE	0xF09E	Set P2P Join profile to join to Autonomous GO
4.8.22 P2P_SET_PROFILE	0xF0A8	Set P2P mode On/Off to firmware
4.8.23 P2P_STOP_FIND	0xF03C	Stop P2P Search
4.8.25 RECONNECT	0x002	Request a reconnection to a BSS

Command Name	Value	Description
4.8.26 SET_BEACON_INT	0x000F	Set the beacon interval for an ad hoc network
4.8.27 SET_BSS_FILTER	0x0009	Inform QCA4002 of network types to receive information about
4.8.28 SET_CHANNEL	0xF042	Sets the channel
4.8.29 SET_CHANNEL_PARAMETERS	0x0011	Configure WLAN channel parameters
4.8.30 SET_FILTERED_PROMISCUOUS_MODE	0xF099	Enable promiscuous mode to listen and forward all traffic to the host
4.8.31 SET_HT_CAP	0xf01E	Set the HT 802.11n capabilities
4.8.32 SET_KEEPALIVE	0x003D	Set a keepalive interval
4.8.33 SET_LISTEN_INT	0x000B	Request a listen interval
4.8.34 SET_PASSPHRASE	0xF048	Set the passphrase
4.8.35 SET_PMK	0xF028	Set the pairwise master key (PMK)
4.8.36 SET_POWER_MODE	0x0012	Set guidelines on trade-off between power utilization
4.8.37 SET_POWER_PARAMS		Configure power parameters
4.8.38 SET_PROBED_SSID	0x000A	Provide list of SSIDs the device should seek
4.8.39 SET_RTS	0x0032	Determine when RTS should be sent
4.8.40 SET_SCAN_PARAMS	0x0008	Set the QCA4002 scan parameters
4.8.41 SET_TX_PWR	0x001B	Sets the current transmit power
4.8.42 SOCKET	0xF08D	Provides socket commands to the IP stack on the QCA4002
4.8.43 START_SCAN	0x0007	Start a long or short channel scan
4.8.44 STORERECALL_CONFIGURE	0xf04E	Enable the store recall command
4.8.45 STORERECALL_RECALL	0xf04F	Restore the saved state after the system wakes up
4.8.46 STORERECALL_HOST_READY	0xf050	Initiates a store state prior to removing the power to the QCA4002
4.8.47 SYNCHRONIZE	0x0004	Force a synchronization point between command and data paths
4.8.48 TARGET_ERROR_REPORT_BITMASK	0x0022	Control ERROR_REPORT events from the QCA4002

Command Name	Value	Description
4.8.49 WPS_START	0xF053	Initiate a WPS enrollee session

4.8.1 ADD_CIPHER_KEY

Synopsis

The host uses this command to add/replace any of four encryption keys on the target. This command is issued after the CONNECT event has been received by the host for all dot11Auth modes except for SHARED_AUTH. When the dot11AuthMode is SHARED_AUTH, then the ADD_CIPHER_KEY command should be issued before the CONNECT command.

Command Parameters: WMI_ADD_CIPHER_KEY_CMD

Type	Name	Comment		
A_UINT8	keyIndex	Index (0...3) of the key to add/replace; uniquely identifies the key		
A_UINT8	keyType	0	NONE_CRYPT	
		2	WEP_CRYPT	
		4	TKIP_CRYPT	
		8	AES_CRYPT	
A_UINT8	keyUsage	Specifies usage parameters of the key when keyType = WEP_CRYPT; all other values are reserved		
		0	PAIRWISE_USAGE	Set if the key is used for unicast traffic only
		1	GROUP_USAGE	Set if the key is used to receive multicast traffic (also set for static WEP keys)
		2	TX_USAGE	Set for the GROUP key used to transmit frames
A_UINT8	keyLength	Length of the key in bytes		
A_UINT8	keyRSC[8]	Key replay sequence counter (RSC) initial value the device should use		
A_UINT8	key[32]	Key material used for this connection		
A_UINT8	keyOpCtrl	Additional key control info; bit[0] = Initialize TSC (default), bit[1] = Initialize RSC		

Type	Name	Comment
A_UINT8	key_macaddr[6]	QCA4002 MAC address filled in the by the application

Reset Value
The four available keys are disabled.

Restrictions
The cipher should correspond to the encryption mode specified in the **CONNECT** command.

See Also
DELETE_CIPHER_KEY

4.8.2 ALLOW_AGGR

Synopsis

Configure TIDs for which aggregation is allowed

Command Parameters: WMI_ALLOW_AGGR_CMD

Type	Name	Comment
A_UINT16	tx_allow_aggr	16-bit mask; bit position indicates TID. 1 indicates aggregation is allowed in uplink.
A_UINT16	rx_allow_aggr	16 bit mask; bit position indicates TID. 1 indicates aggregation is allowed in downlink.

Reset Value

None

Restrictions

None

4.8.3 CONNECT

Synopsis

Command to try to establish the connection based on the previously set profile. Connect control information (connectCtrl) can be used with possible modifiers.

Command Parameters: WMI_CONNECT_CMD

Type	Name	Comment	
A_UINT8	networktype	0x01	INFRA_NETWORK
		0x02	ADHOC_NETWORK

Type	Name	Comment		
		0x04	ADHOC_CREATOR	
A_UINT8	dot11authmode	0x01	OPEN_AUTH	
		0x02	SHARED_AUTH	
A_UINT8	authmode	0x01	NONE_AUTH	
		0x02	WPA_AUTH	
		0x04	WPA2_AUTH	
		0x08	WPA_PSK_AUTH	
		0x10	WPA2_PSK_AUTH	
		0x20	WPA_AUTH_CCKM	
		0x40	WPA2_AUTH_CCKM	
A_UINT8	pairwiseCryptoType	0x01	NONE_CRYPT	
		0x02	WEP_CRYPT	
		0x04	TKIP_CRYPT	
		0x08	AES_CRYPT	
A_UINT8	pairwiseCryptoLen	Length in bytes. Valid when the type is WEP_CRYPT, otherwise this should be 0		
A_UINT8	groupCryptoType	CRYPTO_TYPE, see the fields in pairwiseCryptoType		
A_UINT8	groupCryptoLen	Length in bytes. Valid when the type is WEP_CRYPT, otherwise this should be 0		
A_UINT8	ssidLength	Length of the SSID value		
A_UCHAR	ssid[32]	SSID value of the AP with which the QCA4002 want to connect , in SoftAP mode this is the SSID value with which QCA4002 comes up as SoftAP		
A_UINT16	channel	Channel information with which the QCA4002 connects to AP, in SoftAP mode this is the channel in which the QCA4002 comes up as SoftAP		
A_UINT8	bssid[6]	BSSID value of the AP with which the QCA4002 want to connect , in SoftAP mode this is the BSSID value with which QCA4002 comes up as SoftAP		
A_UINT8	ctrl_flags	WMI_CONNECT_CTRL_FLAGS_BITS		
		0x0001	CONNECT_ASSOC_POLICY_USER	Assoc frames are sent using the policy specified by the CONNECT_SEND-_REASSOC flag

Type	Name	Comment		
		0x0002	CONNECT_SEND_REASSOC	Send Reassoc frame while connecting, otherwise send assoc frames
		0x0004	CONNECT_IGNORE_WPAx-_GROUP_CIPHER	Ignore WPAx group cipher for WPA/WPA2
		0x0008	CONNECT_PROFILE-MATCH_DONE	Ignore any profile check
		0x0010	CONNECT_IGNORE_AAC-_BEACON	Ignore the admission control information in the beacon
		0x0020	CONNECT_CSA_FOLLOW_BSS	Follow BSS/not following CSA; If the BSS sends the STA a channel switch announcement (AP in infrastructure mode) and this flag is set, the STA disconnects in the current channel and rejoins the AP in a new channel. If the flag is reset, the STA disconnects from current BSS. The host must initiate reconnection to a new or old BSS.
		0x0040	CONNECT_DO_WPA_OFFLOAD	Used in the driver supplicant
		0x0080	CONNECT_DO_NOT_DEAUTH	If the STA disconnects from the AP (other than via a DISCONNECT_CMD), do not send the AP a DEAUTH
		0x0100	CONNECT_WPS_FLAG	Set to indicate that the STA must connect to a WPS enabled AP
		0xFFFF		Reset all control flags

Reset Value

None defined

Restrictions

None

4.8.4 DELETE_CIPHER_KEY

Synopsis

The host uses this command to delete a key that was previously added with the ADD_CIPHER_KEY command.

Command Parameters: WMI_DELETE_CIPHER_KEY_CMD

Type	Name	Comment
A_UINT8	keyIndex	Index (0...3) of the key to delete

Reset Value

None

Restrictions

The host should not delete a key that is currently in use by the QCA4002.

See Also

ADD_CIPHER_KEY

4.8.5 DISCONNECT

Synopsis

The host uses this command to disconnect from the currently associated AP or to abort the current connection process. If the host has issued a CONNECT command, it must issue the DISCONNECT command before it issues the next CONNECT command.

Command Parameters

None

Reply Parameters

Disconnect event with reason set to DISCONNECT_CMD

Restrictions

None

See Also

CONNECT

4.8.6 GET_CHANNEL_LIST

Synopsis

Used by the host uses to retrieve the list of channels that can be used by the device while in the current wireless mode and in the current regulatory domain.

Command Parameters

None

Reply Parameters: WMI_CHANNEL_LIST_REPLY

This command generates a channel_list event

Type	Name	Comment
A_UINT8	Reserved	Reserved
A_UINT8	numberOfChannels	Number of channels the reply contains
A_UINT16	channelList[numberOfChannels]	Array of channel frequencies (in MHz)

Reset Values

None defined

Restrictions

The maximum number of channels that can be reported is 32.

4.8.7 GET_KEEPALIVE

Synopsis

The host uses this command to get the configured keepalive interval. If no Tx or Rx activity occurs for the duration of the keepalive interval, the STA must send a NULL data packet to the AP it is connected to.

Command Parameters: WMI_GET_KEEPALIVE_CMD

Type	Name	Comment
A_BOOL	Configured	Shows whether a keepAliveInterval is configured
A_UINT8	keepAliveInterval	keepAliveInterval (in ms)

Reply Parameters

None

Reset Values

None defined

Restrictions

None

See Also

"4.8.32 SET_KEEPALIVE"

4.8.8 GET_PMK

Synopsis

Retrieves the PMK on the firmware.

Command Parameters

None

Reply Parameters: WMI_GET_PMK_REPLY

Type	Name	Comment
A_UINT8	pmk[WMI_PMK_LEN]	The retrieved PMK

Reset Values

None

Restrictions

None

4.8.9 GET_TX_PWR

Synopsis

The QCA4002 maintains counters of significant events to report statistics to the host upon request. This command causes the QCA4002 to send a WMI_REPORT_STATISTICS_EVENTID event with a WMI_TARGET_STATS payload. The payload includes statistics for Rx, Tx, connections, power management, encryption errors, etc. After reporting the statistics the target resets all counters to 0.

Command Parameters

None

Reply Parameters: WMI_TX_PWR_REPLY

Type	Name	Comment
A_UINT16	dBm	The current transmit power specified in dBm

Reset Values

The maximum permitted by the regulatory domain

Restrictions

None

4.8.10 GET_WPS_STATUS

Synopsis

Retrieves the current state of the WPS engine

Command Parameters

None

Reply Parameters: WMI_WPS_STATUS_EVENT

Type	Name	Comment

A_UINT8	wps_status	WPS_STATUS_IDLE = 0 if WPS is idle
		WPS_STATUS_IN_PROGRESSS = 1

A_UINT8	wps_state	Enrollee state is WPS in progress SEND_M1=0, RECV_M2, SEND_M3, RECV_M4, SEND_M5, RECV_M6, SEND_M7, RECV_M8, RECEIVED_M2D, WPS_MSG_DONE, RECV_ACK, WPS_FINISHED, SEND_WSC_NACK

Reset Values

The WPS status is idle

Restrictions

None

4.8.11 P2P_AUTH_GO_NEG

Synopsis

Authorize P2P Device for P2P Connection. Authorized P2P Device connection request is accepted in firmware and starts P2P GO Negotiation with the requested P2P Device. After negotiation P2P_GO_NEG_RESULT event is sent to host.

Command Parameters: WMI_P2P_CONNECT_CMD

Type	Name	Comment
A_UINT16	go_oper_freq	NA
A_UINT8	Dialog_token	0
A_UINT8	Peer_addr[ATH_MAC_LEN]	Peer P2P Device MAC Address
A_UINT8	Own_interface_addr[ATH_MAC_LEN]	NA
A_UINT8	go_dev_dialog_token	NA
P2P_SSID	Peer_go_ssid	NA

A_UINT8	Wps_method	WPS_NOT_READY = 0
		WPS_PIN_LABEL = 1
		WPS_PIN_DISPLAY = 2
		WPS_PIN_KEYPAD = 3
		WPS_PBC = 4

A_UINT8	Dev_capab	NA
A_UINT8	Dev_auth	0 – Authorize ; 1 - Reject
A_UINT8	Go_intent	Go intent value 0 to 15

Reset Value

None defined

Restrictions

None

4.8.12 P2P_CANCEL

Synopsis

This command issued by issue to cancel any current P2P session.

Command Parameters: WMI_P2P_CANCEL_CMD

None

Reset Value

None defined

Restrictions

None

4.8.13 P2P_CONNECT

Synopsis

Host issue P2P Connect to P2P Device which got discovered through P2P_FIND. This command will initiate the P2P GO Negotiation with another P2P Device.
P2P_GO_NEG_RESULT event

Command Parameters: WMI_P2P_FW_CONNECT_CMD

Type	Name	Comment	
A_UINT16	go_oper_freq	NA	
A_UINT8	Dialog_token	0	
A_UINT8	Peer_addr[ATH_MAC_LEN]	Peer P2P Device MAC ADDRESS	
A_UINT8	Own_interface_addr[ATH_MAC_LEN]	Own P2P Device MAC ADDRESS	
A_UINT8	go_dev_dialog_token	0	
P2P_SSID	Peer_go_ssid	NA	
A_UINT8	Wps_method	0	WPS_NOT_READY
		1	WPS_PIN_LABEL
		2	WPS_PIN_DISPLAY
		3	WPS_PIN_KEYPAD
		4	WPS_PBC
A_UINT8	Dev_capab	NA	
A_UINT8	Dev_auth	0	
A_UINT8	Go_intent	Go intent value 0 to 15	

Reset Value

None defined

Restrictions

None

4.8.14 P2P_FIND

Synopsis

Start P2P Device discovery. P2P Device will search for P2P Device in the vicinity. The host must use P2P_NODE_LIST_CMD to have the P2P devices populated.

Command Parameters: WMI_P2P_FIND_CMD

Type	Name	Comment	
A_UINT32	timeout	Timeout in Sec; Default value is 30 sec	
A_UINT8	Type	0	WMI_P2P_FIND_START_WITH_FULL
		1	WMI_P2P_FIND_ONLY_SOCIAL
		2	WMI_P2P_FIND_PROGRESSIVE

Reset Value

None defined

Restrictions

None

4.8.15 P2P_FW_PROV_DISC_REQ

Synopsis

This command issues provisional discovery request to any P2P Device in the P2P Device list. It will result in P2P_PROV_DISC_RESP Event.

Command Parameters: WMI_P2P_PROV_DISC_REQ_CMD

Type	Name	Comment
A_UINT16	wps_method	WPS_CONFIG_PUSHBUTTON 0x0080 WPS_CONFIG_KEYPAD 0x0100 WPS_CONFIG_DISPLAY 0x0008
A_UINT16	Listen_freq	NA

longsys

Type	Name	Comment
A_UINT8	Dialog_token	0
A_UINT8	Peer[ATH_MAC_LEN]	MAC address of PEER P2P Device
A_UINT8	Go_dev_addr[ATH_MAC_LEN]	NA
P2P_SSID	go_oper_ssid	NA

Reset Value

None defined

Restrictions

None

4.8.16 P2P_GET_NODE_LIST

Synopsis

This command will request firmware to deliver populated P2P Devices.

Command Parameters: WMI_P2P_GET_NODE_LIST_CMD

None

Reset Value

None defined

Restrictions

Maximum device populated in firmware is 5.

4.8.17 P2P_GRP_INIT

Synopsis

This command initializes Autonomous GO modules in the firmware. This command will prepare firmware to get ready in Autonomous GO Mode.

WMI_AP_CONFIG_COMMIT_CMD should follow after this command to complete autonomous GO start process.

Command Parameters: WMI_P2P_GRP_INIT_CMD

Type	Name	Comment	
A_UINT8	Persistent_group	0	Non-persistent group
		1	Persistent group
A_UINT8	Group_formation	1	Initialize Auto GO

Reset Value

None defined

Restrictions

None defined

4.8.18 P2P_INVITE_REQ_RSP

Synopsis

This command gives response to P2P_INVITE_REQ event. Host can accept or deny invitation from other P2P GO.

Command Parameters: WMI_P2P_FW_INVITE_REQ_RSP_CMD

Type	Name	Comment	
A_UINT16	Force_freq	NA	
A_UINT8	Status	0	Accept Invitation
		1	Reject invitation
A_UINT8	Dialog_token	0	
A_UINT8	Is_go	0	
A_UINT8	Group_bssid[ATH_MAC_LEN]	MAC address of peer P2P GO	

Reset Value

None defined

Restrictions

None defined

4.8.19 P2P_SET

Synopsis

Host use this command to set following parameter mentioned in the command parameters to own P2P Device.

Command Parameters: WMI_P2P_SET_CMD

Type	Name	Comment

Type	Name	Comment
A_UINT8	Config_id	WMI_P2P_CONFID_LISTEN_CHANNEL=1, WMI_P2P_CONFID_CROSS_CONNECT=2, WMI_P2P_CONFID_SSID_POSTFIX=3, WMI_P2P_CONFID_INTRA_BSS=4, WMI_P2P_CONFID_CONCURRENT_MODE=5, WMI_P2P_CONFID_GO_INTENT=6, WMI_P2P_CONFID_DEV_NAME=7, WMI_P2P_CONFID_P2P_OPMODE=8
union	WMI_P2P_LISTEN_CHANNEL listen_ch; WMI_P2P_SET_CROSS_CONNECT cross_conn; WMI_P2P_SET_SSID_POSTFIX ssid_postfix; WMI_P2P_SET_INTRA_BSS intra_bss; WMI_P2P_SET_CONCURRENT_MODE concurrent_mode; WMI_P2P_SET_GO_INTENT go_intent; WMI_P2P_SET_DEV_NAME device_name; WMI_P2P_SET_MODE mode;	struct { A_UINT8 reg_class; A_UINT8 listen_channel; }WMI_P2P_LISTEN_CHANNEL;
		struct { A_UINT8 flag ; }WMI_P2P_SET_CROSS_CONNECT;
union	WMI_P2P_LISTEN_CHANNEL listen_ch; WMI_P2P_SET_CROSS_CONNECT cross_conn; WMI_P2P_SET_SSID_POSTFIX ssid_postfix; WMI_P2P_SET_INTRA_BSS intra_bss; WMI_P2P_SET_CONCURRENT_MODE concurrent_mode; WMI_P2P_SET_GO_INTENT go_intent; WMI_P2P_SET_DEV_NAME device_name; WMI_P2P_SET_MODE mode;	struct { A_UINT8 reg_class; A_UINT8 listen_channel; }WMI_P2P_LISTEN_CHANNEL;
		struct { A_UINT8 flag ; }WMI_P2P_SET_CROSS_CONNECT;

Type	Name	Comment
		struct { A_UINT8 ssid_postfix[WMI_MAX_SSID_LEN-9] ; A_UINT8 ssid_postfix_len; }WMI_P2P_SET_SSID_POSTFIX;
		struct { A_UINT8 flag ; }WMI_P2P_SET_INTRA_BSS;
		struct { A_UINT8 flag; }WMI_P2P_SET_CONCURRENT_MODE;
		struct { A_UINT8 value; } WMI_P2P_SET_GO_INTENT
		struct { A_UINT8 dev_name[WPS_MAX_DEVNAME_LEN] ; A_UINT8 dev_name_len ; } WMI_P2P_SET_DEV_NAME;
		struct { A_UINT8 p2pmode ; } WMI_P2P_SET_MODE;

Reset Value

None defined

Restrictions

None defined

4.8.20 P2P_SET_CONFIG

Synopsis

Host use this command to set following key configuration before during P2P Device initialization phase.

Command Parameters: WMI_P2P_FW_SET_CONFIG_CMD

Type	Name	Comment	
A_UINT8	go_intent	Go intent range from 0-15	
A_UINT8	country[3]	The following country code are supported: "AR", "AM", "AT", "AU", "AZ", "BH", "BE", "BG", "BN", "BO", "BR", "BY", "BZ", "CH", "CL", "CN", "CY", "CZ", “DB", "DE", "DK", "DO", "EE", "EG", "ES", "FI", "FR", "GB", "GE", "GR", "GT", "HN", "HK", "HR", "HU", "ID", "IE", "IL", "IN", "IR", "IS", "IT", "KP", "KR", "KW", "KZ", "LI", "LT", "LU", "LV", "MA", "MC", "MK", "MY","NL", "NZ", "OM", "PA", "PE", "PH", "PK", "PL", "PT", "QA", "RO", "RU", "SA", "SE", "SG", "SK", "SI", "SV", "SY", "TH", "TT", "TN", "TW", "UA", "US", "UY", "UZ", "VE","YE", "ZA”	
A_UINT8	reg_class	81	
A_UINT8	listen_channel	Social channels 1-6-11	
A_UINT8	op_reg_class	81,115,124,116,117,126,127	
A_UINT8	op_channel	Oper Class	Support Channels
		81	1,2,3,4,5,6,7,8,9,10,11
		115	36,44,48
		124	149,153,157,161
		116	36,44
		117	40,48
		126	149,157

Type	Name	Comment	
		127	153,161
A_UINT32	node_age_to	In seconds	
A_UINT8	max_node_count	1 to 5	

Reset Value

None defined

Restrictions

Node count cannot exceed 5.

4.8.21 P2P_SET_JOIN_PROFILE

Synopsis

Host use this command to set P2P GO profile and provisioning method to firmware to enable firmware to join to a P2P GO.

Command Parameters: WMI_P2P_FW_CONNECT_CMD

Type	Name	Comment	
A_UINT16	go_oper_freq	NA	
A_UINT8	Dialog_token	0	
A_UINT8	Peer_addr[ATH_MAC_LEN]	Peer MAC of P2P Device	
A_UINT8	Own_interface_addr[ATH_MAC_LEN]	Own MAC Address of P2P Device	
A_UINT8	go_dev_dialog_token	0	
P2P_SSID	Peer_go_ssid	NA	
A_UINT8	Wps_method	1	WPS_PIN_LABEL
		2	WPS_PIN_DISPLAY
		3	WPS_PIN_KEYPAD
		4	WPS_PBC
A_UINT8	Dev_capab	NA	
A_UINT8	Dev_auth	NA	
A_UINT8	Go_intent	NA	

Reset Value

None defined

Restrictions

None defined

4.8.22 P2P_SET_PROFILE

Synopsis

This is the first P2P Command to firmware to enable P2P Mode in firmware. All P2P Commands are valid only after P2P Profile is enabled.

Command Parameters: WMI_P2P_SET_PROFILE_CMD

Type	Name	Comment	
A_UINT8	enable	0	Disable P2P mode
		1	Enable P2P mode

Reset Value

None defined

Restrictions

None defined

4.8.23 P2P_STOP_FIND

Synopsis

Host issues this command to stop P2P device discovery.

Command Parameters: WMI_P2P_STOP_FIND_CMD

None

Reset Value

None definedRestrictions

None defined

4.8.24 P2P_LIST_PERSISTENT_NETWORK

Synopsis

The host issues this command to retrieve stored P2P persistent network connections.

Command Parameters: WMI_P2P_LIST_PERSISTENT_NETWORK_CMD

None

Reset Value

None defined

Restrictions

None defined

4.8.25 RECONNECT

Synopsis

This command requests a reconnection to a BSS to which the QCA4002 device was formerly connected.

Command Parameters: WMI_RECONNECT_CMD

Type	Name	Comment
A_UINT16	channel	Provides a hint as to which channel was used for a previous connection
A_UINT8	bssid[6]	If set, indicates which BSSID to connect to

Reset Values

None

Restrictions

None

4.8.26 SET_BEACON_INT

Synopsis

Sets the beacon interval for an ad hoc network. Beacon interval selection may have an impact on power savings. To some degree, a longer interval reduces power consumption but also decreases throughput. A thorough understanding of IEEE 802.11 ad hoc networks is required to use this command effectively.

Command Parameters: WMI_BEACON_INT_CMD

Type	Name	Comment
A_UINT16	beaconInterval	Specifies the beacon interval in TU units (1024 µs)

Reset Values

The default beacon interval is 100 TUs (102.4 ms)

Restrictions

This command can only be issued before the QCA4002 device starts an ad hoc network

4.8.27 SET_BSS_FILTER

Synopsis

The host uses this to inform the QCA4002 device of the types of networks about which it wants to receive information from the "BSSINFO" event. As the device performs either foreground or background scans, it applies the filter and sends "BSSINFO" events only for the networks that pass the filter. If any of the bssFilter or the ieMask filter matches, a BSS Info is sent to the host. The ieMask currently is used as a match for the IEs in the beacons, probe responses, and channel switch action management frame.

Command Parameters: WMI_BSS_FILTER_CMD

Type	Name	Comment	
A_UINT8	bssFilter	Specifies the filter type WMI_BSS_FILTER	
		NONE_BSS_FILTER	0x0, No beacons forwarded
		ALL_BSS_FILTER	All beacons forwarded
		PROFILE_FILTER	Only beacons matching the profile
		ALL_BUT_PROFILE_FILTER	All beacons except the ones matching the profile
		CURRENT_BSS_FILTER	Only beacons matching the current BSS
		ALL_BUT_BSS_FILTER	All beacons except the ones matching the BSS
		PROBED_SSID_FILTER	Beacons matching the probed SSID
		LAST_BSS_FILTER	Marker only
A_UINT8	reserved1	Reserved	
A_UINT16	reserved2	Reserved	
A_UINT32	ieMask	Takes these values:	
		0x01	BSS_ELEMID_CHANSWITCH
		0x02	BSS_ELEMID_ATHEROS

Reply Value

None

Reset Value

BssFilter = NONE_BSS_FILTER (0)

Restrictions

None

4.8.28 SET_CHANNEL

Synopsis

Sets the current channel; use this command to set the channel when sending a raw frame

Command Parameters: WMI_CHANNEL_CMD

Type	Name	Comment	
A_UINT16	Channel	Center frequency of the channel. The channel frequencies in the 2.4GHz bands are:	
		2412	FREQ_1
		2417	FREQ_2
		2422	FREQ_3
		2427	FREQ_4
		2432	FREQ_5
		2437	FREQ_6
		2442	FREQ_7
		2447	FREQ_8
		2452	FREQ_9
		2457	FREQ_10
		2462	FREQ_11
		2457	FREQ_12 / Europe, Japan
		2472	FREQ_13 / Europe, Japan
		2484	FREQ_14 / Japan

Reset Values

None

Restrictions

None

4.8.29 SET_CHANNEL_PARAMETERS

Synopsis

Configures various WLAN parameters related to channels, sets the wireless mode, and can restrict the QCA4002 device to a subset of available channels. The list of available channels varies depending on the wireless mode and the regulatory domain. The device never operates on a channel outside of its regulatory domain.

This command causes the device to scan the list of channels.

Command Parameters: WMI_CHANNEL_PARAMS_CMD

Type	Name	Comment	
A_UINT8	reserved1	Reserved	
A_UINT8	scanParam	Set whether to enable auto scan	
		0	Do not scan after executing this command
		1	Auto scan after executing this command
A_UINT8	phyMode	WMI_PHY_MODE	
		0x1	WMI_11A_MODE
		0x2	WMI_11G_MODE
		0x3	WMI_11AG_MODE
		0x4	WMI_11B_MODE
		0x5	WMI_11GONLY_MODE
A_UINT8	numChannels	Number of channels in the channel array that follows. If = 0, then the device uses all of the channels permitted by the regulatory domain and by the specified phyMode.	
A_UINT16	channelList[numChannels]	Array listing the subset of channels (expressed as frequencies in MHz) the host wants the device to use. Any channel not permitted by the specified phyMode or by the specified regulatory domain is ignored by the device.	

Reset Values

- phyMode: WMI_11G_MODE
- channelList: Defaults to all channels permitted by the current regulatory domain.

Restrictions

This command, if issued, should be issued soon after reset and prior to the first connection. This command should only be issued in the DISCONNECTED state.

The QCA4002 only supports the 2.4 GHz band.

4.8.30 SET_FILTERED_PROMISCUOUS_MODE

Synopsis

Promiscuous mode allows sniffing of packets in the air. Due to the potential that the amount of traffic at any time may overwhelm the bus interface, a filtering mechanism is provided that limits how much traffic is actually allowed across the bus interface.

Command Parameters: WMI_SET_FILTERED_PROMISCUOUS_MODE_CMD

Type	Name	Comment	
A_UINT8	enable	Enables or disables promiscuous mode	
A_UINT8	Filters	Filter is a bitmask to optionally control which packets are sent to the host. The filter set to DST applies to the address1 field in the 802.11 MAC headers, and SRC applies to the address2 field. If a bit is set, then the corresponding address field in the MAC header is matched with the provided source or destination address. Only those frames having a match are forwarded to the host. The bit positions for the filters are defined as:	
		0x0	WMI_PROM_FILTER_DST_ADDR
		0x01	WMI_PROM_FILTER_SRC_ADDR
A_UINT8	srcAddr[6]	Address2 field in the 802.11 header to allow sending to the host if matching this value	
A_UINT8	dstAddr[6]	Address1 field in the 802.11 header to allow sending to the host if matching this value	

Reset Values

Disabled with no filters set

Restrictions

Promiscuous mode is only intended to be used in isolation. Attempting 802.11 commands when this mode is enabled will produce unknown results.

4.8.31 SET_HT_CAP

Synopsis

Controls and configures HT (802.11n) capabilities. Values and use of SGI, 40-MHz operation, 40-MHz intolerance, and the max AMPDU exponent can be controlled via this command.

Command Parameters: WMI_SET_HT_CAP_CMD

Type	Name	Comment
A_UINT8	band	Specifies for which band the following parameters should be applied.
A_UINT8	enable	Enables or disables HT support.
A_UINT8	chan_width_40M_supported	Indicates whether 40 MHz operation is supported. The value will be used to populate the HT Information Element as it appears in Association frames, Probe frames and Beacons
A_UINT8	short_GI_20MHz	Indicates whether SGI is supported for 20 MHz operation. The value will be used to populate the HT Information Element as it appears in Association frames, Probe frames, and Beacons.
A_UINT8	short_GI_40MHz	Indicates whether SGI is supported for 40 MHz operation. The value will be used to populate the HT Information Element as it appears in Association frames, Probe frames, and Beacons.
A_UINT8	intolerance_40MHz	Indicates whether or not this device is intolerant of 40 MHz operation. Used to populate the HT information element as it appears in association frames, probe frames, and beacons.
A_UINT8	max_ampdu_len_exp	Indicates the maximum number of bytes that can be successfully received in an AMPDU. Used to populate the HT information element as it appears in association frames, probe frames, and beacons.

Reset Value

- enable = 1 <HT enabled>
- 40MHz_support = 0 <not supported>
- SGI_20MHz = 1 <supported>
- SGI_40MHz=1 <supported>
- 40MHz_intolerant = 0 <not intolerant>
- Max_AMPDU_EXP = 1 <16384 bytes per AMPDU>

Restrictions

Set prior to invoking any WLAN operations; MAX_AMPDU_LEN_EXP should not exceed 1

4.8.32 SET_KEEPALIVE

Synopsis

The host uses this command to set a keepalive interval. If there is no transmission or reception activity for the duration of the keepalive interval, the STA must send a NULL data packet to the AP it is connected to.

Command Parameters: WMI_SET_KEEPALIVE_CMD

Type	Name	Comment
A_UINT8	keepAliveInterval	Keepalive interval (in ms)

Command Values

None defined

Reset Value

None defined

Restrictions

None

See Also

GET_KEEPALIVE

4.8.33 SET_LISTEN_INT

Synopsis

The host uses this command to request a listen interval, which determines how often the QCA4002 device should wake up and listen for traffic. The listen interval can be set by the TUs or by the number of beacons. The device may not be able to comply with the request (e.g., if the beacon interval is greater than the requested listen interval, the device sets the listen interval to the beacon interval). The actual listen interval used by the device is available in the CONNECT event.

Command Parameters: WMI_LISTEN_INT_CMD

Type	Name	Comment
A_UINT16	listenInterval	Specifies the listen interval in Kμs (1024 μs), ranging from 100 to 1000
A_UINT16	listenBeacons	Specifies the listen interval in beacons, ranging from 1 to 50

Command Values

None

Reset Values

The device sets the listen interval equal to the beacon interval of the AP it associates to.

Restrictions

None

4.8.34 SET_PASSPHRASE

Synopsis

The host uses this command to set the passphrase.

Command Parameters: WMI_SET_PASSPHRASE_CMD

Type	Name	Comment
A_UCHAR	ssid[32]	The SSID to be used for the passphrase calculation
A_UINT8	passphrase[WMI_PASSPHRASE_LEN=64]	The passphrase generation value
A_UINT8	ssid_len	Actual length of the SSID
A_UINT8	passphrase_len	Actual length of the passphrase

Reset Values

None defined

Restrictions

None

4.8.35 SET_PMK

Synopsis

The host uses this command to set the pairwise master key (PMK).

Command Parameters: WMI_SET_PMK_CMD

Type	Name	Comment
A_UINT8	pmk[WMI_PMK_LEN]	The PMK to be set

Reset Values

None defined

Restrictions

None

See Also

"4.8.8 GET_PMK"

4.8.36 SET_POWER_MODE

Synopsis

The host uses this command to provide the QCA4002 device with guidelines on the desired trade-off between power utilization and performance.

- In normal power mode, the device enters a sleep state if they have nothing to do, which conserves power but may cost performance as it can take up to 2 ms to resume operation after leaving sleep state.
- In maximum performance mode, the device never enters sleep state, thus no time is spent waking up, resulting in higher power consumption and better performance.

Command Parameters: WMI_SET_POWER_MODE_CMD

Type	Name	Comment
A_UINT8	powerMode	WMI_POWER_MODE REC_POWER = 1 (Recommended setting) Tries to conserve power without sacrificing performance MAX_PERF_POWER = 2 Setting that maximizes performance at the expense of power All other values are reserved

Reset Values

powerMode is REC_POWER

Restrictions

None

4.8.37 SET_POWER_PARAMS

Synopsis

The host uses this command to configure power parameters

Command Parameters: WMI_SET_POWER_PARAMS_CMD

Type	Name	Comment		
A_UINT16	idle_period	Length of time (in ms) the QCA4002 device remains awake after frame Rx/Tx before going to SLEEP state		
A_UINT16	pspoll_number	The number of power save Poll (PS-poll) messages the device should send before notifying the AP it is awake		
A_UINT16	dtim_policy	WMI_DTIM_POLICY		
		1	IGNORE_DTIM	The device does not listen to any content after beacon (CAB) traffic
		2	NORMAL_DTIM	DTIM period follows the listen interval (e.g., if the listen interval is 4 and the DTIM period is 2, the device wakes up every fourth beacon)
		3	STICK_DTIM	Device attempt to receive all CAB traffic (e.g., if the DTIM period is 2 and the listen interval is 4, the device wakes up every second beacon)
		4	AUTO_DTIM	Device decides
A_UINT16	tx_wakeup_po	WMI_TX_WAKEUP_POLICY_UPON_SLEEP		

Type	Name	Comment		
	licy	1	TX_WAKEUP_UPON_SLEEP	Indicate awake to AP upon uplink traffic. Number of uplink frames in a beacon interval to transition to awake is determined by NUM_TX_TO_WAKEUP
		2	TX_DONT_WAKEUP_UPON_SLEEP	Never transition
A_UINT16	num_tx_to_wakeup			
A_UINT16	ps_fail_event_policy	POWER_SAVE_FAIL_EVENT_POLICY		
		1	SEND_POWER_SAVE_FAIL_EVENT_ALWAYS	Any null frame with PM=1 Tx failure is notified via a WMI_TARGET_PM_ERR_FAIL event
		2	IGNORE_POWER_SAVE_FAIL_EVENT_DURING_SCAN	Ignore Null frame with PM=1 Tx failures during scan

Reset Values

- idle_period = 200 ms
- pspoll_number = 1
- dtim_policy = STICK_DTIM
- tx_wakeup_policy = TX_WAKEUP_UPON_SLEEP
- num_tx_to_wakeup = 1
- ps_fail_event_policy = SEND_POWER_SAVE_FAIL_EVENT_ALWAYS

Restrictions

None

4.8.38 SET_PROBED_SSID

Synopsis

The QCA4002 keeps a list of up to MAX_PROBED_SSID_INDEX + 1 SSIDs that the device should actively look for. The first entry is reserved for SSID provided by CONNECT CMD. Others entries are set by this command. By default, the device actively looks for only the SSID specified in the 4.9.4 CONNECT command, and only when the regulatory domain allows active probing. With this command, specified SSIDs are probed for, even if they are hidden.

Command Parameters: WMI_PROBED_SSID_CMD

Type	Name	Comment		
A_UINT8	entryIndex	A number from 0 to MAX_PROBED_SSID_INDEX indicating the active SSID table entry index for this command (if the specified entry index already has an SSID, the SSID specified in this command replaces it). For example, etnryIndex = 0 modifies the second entry in the list.		
A_UINT8	flag	WMI_SSID_FLAG indicates the current entry in the active SSID table		
		0	DISABLE_SSID_FLAG	Disables entry
		1	SPECIFIC_SSID_FLAG	Probes specified SSID
		2	ANY_SSID_FLAG	Probes for any SSID
A_UINT8	ssidLength	Length of the specified SSID in bytes. If = 0, the entry corresponding to the index is erased		
A_UINT8	ssid[32]	SSID string actively probed for when permitted by the regulatory domain		

Reset Value

The entries are unused.

Restrictions

None

4.8.39 SET_RTS

Synopsis

Decides when RTS should be sent.

Command Parameters: WMI_SET_RTS_CMD

Type	Name	Comment
A_UINT16	threshold	Command parameter threshold in bytes. An RTS is sent if the data length is more than this threshold. The default is to NOT send RTS.

Reset Value

Not to send RTS.

Restrictions

None

4.8.40 SET_SCAN_PARAMS

Synopsis

The host uses this command to set the QCA4002 scan parameters, including the duty cycle for both foreground and background scanning. Foreground scanning takes place when the QCA4002 device is not connected, and discovers all available wireless networks to find the best BSS to join. Background scanning takes place when the device is already connected to a network and scans for potential roaming candidates and maintains them in order of best to worst. A second priority of background scanning is to find new wireless networks.

The device initiates a scan when necessary. For example, a foreground scan is always started on receipt of a CONNECT command or when the device cannot find a BSS to connect to. Foreground scanning is disabled by default until receipt of a CONNECT command. Background scanning is enabled by default and occurs every

60 seconds after the device is connected. The device implements a binary back off interval for foreground scanning when it enters the DISCONNECTED state after losing connectivity with an AP or when a CONNECT command is received. The first interval is ForegroundScanStartPeriod, which doubles after each scan until the interval reaches ForegroundScanEndPeriod.

If the host terminates a connection with DISCONNECT, the foreground scan period is ForegroundScanEndPeriod. All scan intervals are measured from the time a full scan ends to the time the next full scan starts. The host starts a scan by issuing a SOCKET command.

Command Parameters: WMI_SCAN_PARAMS_CMD

Type	Name	Comment

Type	Name	Comment		
A_UINT16	fg_start_period	First interval (seconds) used by the device when it disconnects from an AP or receives a connect command, specified in seconds (0–65535). If = 0, the device uses the reset value. If = 65535, the device disables foreground scanning.		
A_UINT16	fg_end_period	The maximum interval (seconds) the device waits between foreground scans specified in seconds (from ForegroundScanStartPeriod to 65535). If = 0, the device uses the reset value.		
A_UINT16	bg_period	The period of background scan specified in seconds (0–65535). By default, it is set to the reset value of 60 seconds. If 0 or 65535 are specified, the device disables background scanning.		
A_UINT16	maxact_chdwell_time	The period of time the device stays on a particular channel while active scanning, specified in msec (10–65535). If the special value of 0 is specified, the device uses the reset value.		
A_UINT16	pas_chdwell_time	The period of time the device remains on a particular channel while passive scanning. It is specified in ms (10–65535). If the special value of 0 is specified, the device uses the reset value.		
A_UINT8	shortScanRatio	Number of short scans to perform for each long scan.		
A_UINT8	scanCtrlFlags	WMI_SCAN_CTRL_FLAGS_BITS		
		0x01	CONNECT_SCAN_CTRL_FLAGS	Set whether the QCA4002 can scan using the connect command, else no probe request is sent during connect
		0x02	SCAN_CONNECTED_CTRL_FLAGS	Set whether the QCA4002 can scan for the SSID it will connect to or is already connected to
		0x04	ACTIVE_SCAN_CTRL_FLAGS	Set whether active scan is enabled for the SSID

Type	Name	Comment			
		0x08	ROAM_SCAN_CTRL_FLAGS	Set whether roam scan is enabled when BMISS and a low RSSI event happens	
		0x10	REPORT_BSSINFO_CTRL_FLAGS	Set whether follows customer BSSINFO reporting rule	
		0x20	ENABLE_AUTO_CTRL_FLAGS	If not set, the target does not scan automatically to find an AP after a disconnect event.	
		0x40	ENABLE_SCAN_ABORT_EVENT	A scan complete event with canceled status is generated when a scan is preempted before it completes. ScanCtrlFlags =	
				0x0	Disable all scan control flags
				0xFF	Leave the current settings
A_UINT16	minact_chdwell_time	Specified in ms			
A_UINT16	maxact_scan_per_ssid	Maximum active scans per SSID			
A_UINT32	max_dfsch_act_time	Maximum time a DFS channel can stay active before being marked passive, in ms			

Reset Values

- ForegroundScanStartPeriod = 1 sec
- ForegroundScanEndPeriod = 60 sec
- BackgroundScanPeriod = 60 sec
- ActiveChannelDwellTime = 105 ms

Restrictions

None

4.8.41 SET_TX_PWR

Synopsis

The host uses this command to specify the Tx power level of the QCA4002; it cannot be used to exceed the power limit permitted by the regulatory domain. The maximum output power is limited in the chip to 31.5 dBm; the range is 0 – 31.5 dBm..

Command Parameters: WMI_SET_TX_PWR_CMD

Type	Name	Comment
A_UINT8	dbM	The desired transmit power in dbM

Reset Value

The maximum permitted by the regulatory domain.

Restrictions

None

See Also

GET_TX_PWR

4.8.42 SOCKET

Synopsis

The socket command is the API for all functionality provided by the on Chip IP stack on the QCA4002. The files atheros_stack_offload.h and api_stack_offload.c in the reference implementation provides an example of using this interface.

NOTE: This command is only present in the QCA4002. Refer to chapter 2 and 6 for additional information on using the IP Stack.

Command Parameters: WMI_SOCKET_CMD

Type	Name	Comment
A_UINT32	commandType	Supported commands are defined in the SOCKET_CMDS enum; supported values are:

Type	Name	Comment	
		SOCK_OPEN	Open a socket Domain can specify IPv4 (INET) or IPv6 (INET6). A TCP connection is opened specifying a STREAM type. A UDP command by a DGRAM type.
		SOCK_CLOSE	Close existing socket
		SOCK_CONNECT	Connect to a peer The connect command is used to establish a connection to the remote peer; it is not mandatory for a UDP connection
		SOCK_BIND	Bind to interface The bind command is used to tie a UDP socket to an interface (local IP address and port)
		SOCK_LISTEN	Listen on socket This command waits for an incoming TCP connection. The backlog parameter defines the maximum length for the queue of pending connections (not maximum open connections).
		SOCK_ACCEPT	Accept incoming connection The accept command is used by the remote peer to accept a connection request for a TCP socket. The driver is expected to block to get the requesting socket's connection information and create a socket context to refer to this connection. These 3 commands handle both IPv4 and IPv6.
		SOCK_SETSOCKOPT	Set specified socket options
		SOCK_GETSOCKOPT	Get socket options
		SOCK_IPCONFIG	Set static IP information, or get current IP configuration
		SOCK_IP6CONFIG	
		SOCK_PING	Send an IP ping

Type	Name	Comment	
		SOCK_PING6	
		SOCK_STACK_INIT	Command to initialize IP stack on the target
		SOCK_STACK_MISC	Used to exchange misc. information, for example, reassembly
		SOCK_IP_SET_IP6_STATUS	Enable /Disable the IPv6 mode
		SOCK_IPCONFIG_DHCP_POOL	Specify the DHCP Pool size and address
		SOCK_IP6CONFIG_ROUTER_PREFIX	Set IPv6 Router Prefix
		SOCK_IP_DHCP_RELEASE	Release the DHCP IP Address
		SOCK_IP_SET_TCP_EXP_BACKOFF_RETRY	Set the no of TCP Retransmissions. By default it is 12.
		SOCK_IP_SET_TCP_RX_BUF	Set the no of RX Buffers for TCP
		SOCK_HTTP_SERVER	Set HTTP Server module status enable/disable
		SOCK_HTTP_SERVER_CMD	Command to issue HTTP GET and POST
		SOCK_DNC_CMD	Set the DNS resolver commands .It can be either one of the below commands GETHOSTBYNAME (To get a IPv4 address) GETHOSTBYNAME2 (To get a ipv6 address) RESOLVEHOSTNAME (To resolve domain name. Can return either ipv4 or ipv6 address based on the DOMAIN TYPE expected.
		SOCK_DNC_ENABLE	Set the DNS client module to enable/disable
		SOCK_DNS_SERVER_ADDR	Set the DNS Server Address

Type	Name	Comment	
		Size in bytes of the data that follows. Each command has its own command structure.	
		Command structure	

		SOCK_OPEN	typedef struct sock_open {
			A_UINT32 domain; //ATH_AF_INET, ATH_AF_INET6
			A_UINT32 type;
			//SOCK_STREAM_TYPE, SOCK_DGRAM_TYPE
			A_UINT32 protocol; // 0
			} SOCK_OPEN_T;

		SOCK_CLOSE	typedef struct sock_close {
			A_UINT32 handle; //socket handle
			} _CLOSE_T;

		SOCK_CONNECT	
		SOCK_BIND	

		SOCK_LISTEN	typedef struct sock_listen {
			A_UINT32 handle; //Socket handle
			A_UINT32 backlog; //Max length of queue of backlog connections
			} SOCK_LISTEN_T;

Type	Name	Comment	

		SOCK_ACCEPT	typedef struct sock_connect {
			A_UINT32 handle; //socket handle
			union
			{
			SOCKADDR_T name;
			SOCKADDR_6_T name6;
			}addr; // IP Address and port
			A_UINT16 length; //socket address length
			} SOCK_CONNECT_T, SOCK_BIND_T,
			SOCK_ACCEPT_T;
			Where
			typedef struct sockaddr {
			A_UINT16 sin_port; //Port number
			A_UINT16 sin_family; //ATH_AF_INET
			A_UINT32 sin_addr; //IPv4 Address
			} SOCKADDR_T; // Is used for IPv4
			typedef struct sockaddr_6
			{
			A_UINT16 sin6_family; //
			ATH_AF_INET6
			A_UINT16 sin6_port; //
			transport layer port #
			A_UINT32 sin6_flowinfo; // IPv6 flow
			information
			IP6_ADDR_T sin6_addr; // IPv6
			address
			A_UINT32 sin6_scope_id; // set of
			interfaces for a scope
			} SOCKADDR_6_T; // Is used for an IPv6 address

		SOCK_SETSOCKOPT	The following options are supported: IP_ADD_MEMBERSHIP=12 // ip_mreq; add IP

Type	Name	Comment	

		SOCK_GETSOCKOPT	group membership
			IP_DROP_MEMBERSHIP=13 // ip_mreq; drop IP gp. membership
			IPV6_JOIN_GROUP=83 // ipv6_mreq; join MC group
			IPV6_LEAVE_GROUP=84 // ipv6_mreq; leave MC group
			
			typedef struct sock_setopt
			{
			A_UINT32 handle; //socket handle
			A_UINT32 level;
			// Option level (ATH_IPPROTO_IP,
			// ATH_IPPROTO_UDP, ATH_IPPROTO_TCP...)
			A_UINT32 optname; //Choose from list above
			A_UINT32 optlen; // option length
			A_UINT8 optval[1]; // option value
			} SOCK_OPT_T;

		SOCK_IPCONFIG	Command to get the IPv4 configuration parameters.

Type	Name	Comment	

		SOCK_IP6CONFIG	SOCK_IP6CONFIG: Command to get the IPv6 configuration parameters.
			IPv6 only supports a mode of IPCFG_QUERY.
			typedef struct ipconfig {
			A_UINT32 mode;
			// IPCFG_QUERY=0, Retrieves the IP parameters
			// IPCFG_STATIC=1, Set static IP parameters
			// IPCFG_DHCP=2, Run DHCP client
			A_UINT32 ipv4; // IPv4 address
			A_UINT32 subnetMask; // IPv4 subnet mask
			A_UINT32 gateway4; // IPv4 gateway
			IP6_ADDR_T ipv6LinkAddr; // IPv6 Link Local address
			IP6_ADDR_T ipv6GlobalAddr; // IPv6 Global address
			IP6_ADDR_T ipv6DefGw; // IPv6 Default Gateway
			IP6_ADDR_T ipv6LinkAddrExtd; // IPv6 Link Local address
			A_INT32 LinkPrefix;
			A_INT32 GlbPrefix;
			A_INT32 DefGwPrefix;
			A_INT32 GlbPrefixExtd;
			} IPCONFIG_T;

		SOCK_PING	Sends an IPv4 ping.
			typedef struct ping {
			A_UINT32 ip_addr; // Destination IPv4 address
			A_UINT32 size; // size of packet to send
			} PING_T;

		SOCK_PING6	typedef struct ping6 {
			A_UINT8 ip6addr[16]; //Destination IPv6 address
			A_UINT32 size; // size of packet to send
			} PING_6_T;

Type	Name	Comment	

		SOCK_STACK_INIT	typedef struct ath_sock_stack_init {
			A_UINT8 stack_enabled;
			// Flag to indicate if stack should be enabled in the target
			A_UINT8 num_sockets;
			// number of sockets supported by the host
			A_UINT8 num_buffers;
			//Number of RX buffers supported by host
			A_UINT8 reserved;
			} ATH_STACK_INIT;

		SOCK_IP_SET_IP6_STATUS	typedef PREPACK struct sock_ipv6_status {
			A_UINT16 ipv6_status FIELD_PACKED;
			// the status field indicates whether to enable /disable IPv6 by setting this field to 1 or 0 respectively
			}POSTPACK SOCK_IPv6_STATUS_T;

		SOCK_IPCONFIG_DHCP_POOL	typedef PREPACK struct ipconfigdhcppool {
			A_UINT32 startaddr FIELD_PACKED;
			//this is used to specify the 4 byte IPv4 start address for DHCP pool
			A_UINT32 endaddr FIELD_PACKED;
			//this is used to specify the 4 byte IPv4 end address for DHCP pool
			A_INT32 leasetime FIELD_PACKED;
			//this is used to specify the valid lease time for DHCP pool we provide
			}POSTPACK IPCONFIG_DHCP_POOL_T;

Type	Name	Comment	

		SOCK_IP6CONFIG_ROUTER_PREFIX	typedef PREPACK struct ip6config_router_prefix {
			 A_UINT8 v6addr[16] FIELD_PACKED;
			 //this will specify the IPv6 start address
			 A_INT32 prefixlen FIELD_PACKED;
			 //this will specify the prefix len for the IPv6 address
			 A_INT32 prefix_lifetime FIELD_PACKED;
			 //this parameter will specify the lifetime for the prefix used
			 A_INT32 valid_lifetime FIELD_PACKED;
			 // this will specify the lifetime for the IPv6 address used
			 }POSTPACK IP6CONFIG_ROUTER_PREFIX_T;

		SOCK_IP_DHCP_RELEASE	typedef PREPACK struct sock_ip_dhcp_release {
			 A_UINT16 ifIndex FIELD_PACKED;
			 //this parameter will indicate which interface has to release the IPv4 address , currently only 1 interface supported , DHCP release is supported for IPv4 only
			 }POSTPACK SOCK_IP_DHCP_RELEASE_T;

		SOCK_HTTP_SERVER	typedef PREPACK struct sock_ip_http_server {
			 A_INT32 enable FIELD_PACKED;
			 This parameter enables/disables the HTTP Server feature at run time
			 }POSTPACK HTTP_SERVER_ENABLE_T

Type	Name	Comment	

		SOCK_HTTP_SERVER_CMD	typedef PREPACK struct http_server_command
			{
			A_INT32 command FIELD_PACKED;
			A_UINT8 pagename[32] FIELD_PACKED;
			A_UINT32 objname[32] FIELD_PACKED;
			A_UINT32 objlen FIELD_PACKED;
			A_UINT8 value[32] FIELD_PACKED;
			The command can be either GET or POST.
			Pagename specified the page on which the operation has to be done. In case of POST the values can be updated by specifying the object name , length and value
			} POSTPACK HTTP_SERVER_COMMAND_T

		SOCK_DNC_CMD	typedef PREPACK struct dnccfgcmd {
			A_UINT32 mode FIELD_PACKED;
			A_UINT32 domain FIELD_PACKED;
			A_UINT8 ahostname[36] FIELD_PACKED;
			//This structure has to be filled by the application for resolving a domain name.
			Mode can be either of these values
			a)GETHOSTBYNAME,b)GETHOSTBYNAME2
			c)RESOLVEHOSTNAME.
			The domain can be either 1 (ipv4) and 2 (ipv6).
			The ahostname is used to specify the name for which IP Address is to be resolved.

		SOCK_DNC_ENABLE	typedef PREPACK struct sock_dns_client {
			A_INT32 command FIELD_PACKED;
			//This parameter will enable /disable the DNS Client module
			}POSTPACK SOCK_IP_DNS_CLIENT_T

		SOCK_DNS__SRVR_CFG_ADDR	typedef PREPACK struct sock_ip_dns_config_server_addr {
			IPV46ADDR addr FIELD_PACKED;
			// This parameter will configure either ipv4 or ipv6 dns server address
			}POSTPACK SOCK_IP_CFG_DNS_SRVR_ADDR

Type	Name	Comment	

		SOCK_HTTPC	typedef PREPACK struct httpc_command {
			A_UINT32 command FIELD_PACKED;
			A_UINT8 url[32] FIELD_PACKED;
			A_UINT8 data[64] FIELD_APCKED;
			}POSTPACK SOCK_HTTPC_T;
			This structure configures the HTTP Client for perfroming GET and POST operations

		SOCK_DNS_LOCAL_DOMAIN	typedef PREPACK struct sock_ip_dns_local_domain {
			WMI_SOCKET_CMD wmi_cmd FIELD_PACKED;
			A_CHAR domain_name[33] FIELD_PACKED; This parameter configures the local domain
			}POSTPACK SOCK_IP_CFG_DNS_LOCAL_DOMAIN;

		SOCK_IP_HOST_NAME	typedef PREPACK struct sock_ip_dns_hostname {
			A_CHAR domain_name[33] FIELD_PACKED; This paranmeter configures the host name
			}POSTPACK SOCK_IP_CFG_HOST_NAME;

		SOCK_IP_DNS	typedef PREPACK struct sock_ip_dns {
			A_INT32 command FIELD_PACKED;
			A_CHAR domain_name[36] FIELD_PACKED;
			IP46ADDR addr FIELD_PACKED;
			}POSTPACK SOCK_IP_DNS_T
			/* This structure configures the DNS database */

Reset Value

None

Restrictions

This command is only supported on the QCA4002.

4.8.43 START_SCAN

Synopsis

The host uses this command to start a long or short channel scan. All future scans are relative to the time the QCA4002 device processes this command. The device performs a channel scan on receipt of this command, even if a scan was already in progress. The host uses this command when it wishes to refresh its cached database of wireless networks. The isLegacy field will be removed (0 for now) because it is achieved by setting CONNECT_PROFILE_MATCH_DONE in the CONNECT command.

Command Parameters: WMI_START_SCAN_CMD

Type	Name	Comment		
A_BOOL	forceFgScan	Forces a foreground scan		
A_BOOL	isLegacy	For legacy Cisco compatibility		
A_UINT32	homeDwellTime	Maximum duration in the home channel (milliseconds)		
A_UINT32	forceScanInterval	Time interval between scans (milliseconds)		
A_UINT8	scanType	WMI_SCAN_TYPE		
		0x0	WMI_LONG_SCAN	Requests a full scan
		0x1	WMI_SHORT_SCAN	Requests a short scan
A_UINT8	numChannels	Number of channels that follow		
A_UINT16	channelList[numChannels]	Channels in MHz		

Reset Value

Disable forcing foreground scan

Restrictions

isLegacy field will no longer be supported (pass as 0 for now)

4.8.44 STORERECALL_CONFIGURE

Synopsis

This command is used to enable the store recall command. It is also used to configure where the saved data is intended to be sent to.

Command Parameters: WMI_STORERECALL_CONFIGURE_CMD

Type	Name	Comment
A_UINT8	enable	Enable the store recall mode
A_UINT8	recipient	Intended recipient of the stored data. Currently set to STRRCL_RECIPIENT_HOST = 0x1

Reset Value

Disabled.

Restrictions

The store-recall mechanism needs to be enabled prior to the connection.

4.8.45 STORERECALL_RECALL

Synopsis

This command is used to restore the saved state after the system has woken up.

Command Parameters: WMI_SET_STORERECALL_RECALL_CMD

Type	Name	Comment
A_UINT32	length	Number of bytes of data to follow
A_UINT8	data[length]	Raw data stream of state information that was stored on the host from a prior STORERECALL_HOST_READY command

Reset Value

None

Restrictions

None

See Also

"4.8.46 STORERECALL_HOST_READY"

4.8.46 STORERECALL_HOST_READY

Synopsis

This command is used to initiate a store state prior to removing the power to the QCA4002. The result of this command will generate a WMI_STORERECALL_STORE_EVENT which will return the saved data, as well as specify the actual time the caller needs to sleep to get the effect of the requested sleep duration specified in this command.

Command Parameters: WMI_STORERECALL_HOST_READY_CMD

Type	Name	Comment
A_UINT32	sleep_msec	Duration of time the host would like to sleep

Type	Name	Comment
A_UINT8	store_aftertx_empty	Specified that the target needs to wait for all pending transmit packets to be transmitted
A_UINT8	store_after_fresh_beacon_rx	Request to stay awake till the receipt of the next beacon (to synchronize the clock)

Reset Value

None

Restrictions

None

4.8.47 SYNCHRONIZE

Synopsis

The host uses this command to force a synchronization point between the command and data paths.

Command Parameters: WMI_SYNC_CMD

Type	Name	Comment
A_UINT8	dataSyncMap	A bitmap of endpoints that need to be synchronized

Reset Values

None

Restrictions

None

4.8.48 TARGET_ERROR_REPORT_BITMASK

Synopsis

Allows the host to control “ERROR_REPORT” events from the QCA4002 device.

- If error reporting is disabled for an error type, a count of errors of that type is maintained by the device.
- If error reporting is enabled for an error type, an “ERROR_REPORT” event is sent when an error occurs and the error report bit is cleared.

Command Parameters: WMI_TARGET_ERROR_REPORT_BITMASK

Type	Name	Comment

Type	Name	Comment		
A_UINT32	bitmask	The set of WMI_TARGET_ERROR_VAL error types enabled for reporting		
		0x00000001	WMI_TARGET_PM_ERR_FAIL	Power save fails (only two cases): ▪ Retry out of null function/QoS null function to associated AP for PS indication ▪ Host changes the PS setting when STA is off home channel
		0x00000002	WMI_TARGET_KEY_NOT_FOUND	No cipher key
		0x00000004	WMI_TARGET_DECRYPTION_ERR	Decryption error
		0x00000008	WMI_TARGET_BMISS	Beacon miss
		0x00000010	WMI_PSDISABLE_NODE_JOIN	A non-PS-enabled STA joined the PS-enabled network
		0x00000020	WMI_TARGET_COM_ERR	HTC error
		0x00000040	WMI_TARGET_FATAL_ERR	Fatal error

Reset Values

Bitmask is 0, and all error reporting is disabled.

Restrictions

None

4.8.49 WPS_START

Synopsis

This command is used to initiate a WPS enrollee session.

Command Parameters: WMI_WPS_START_CMD

Type	Name	Comment		
WPS_SCAN_LIST_ENTRY	ssid_info	A_UINT8	ssid[32]	Optional SSID restriction
		A_UINT8	macaddress[6]	Optional MAC restriction
		A_UINT16	channel	Optional channel restriction
A_UINT8	config_mode	WPS_MODE		
		0x1	WPS_PIN_MODE	Use PIN mode
		0x2	WPS_PBC_MODE	Use push button mode
WPS_PIN	wps_pin	A_UINT8 pin[9] specifies the PIN for the PIN mode		
A_UINT8	Timeout	Timeout (seconds) for the WPS session		

Reset Value

None

Restrictions

None

4.8.50 GET_BITRATE

Synopsis

This command is used to get the last transmitted bit rate from the chip

Command Parameters: WMI_GET_BITRATE_CMDID

Reset Value

None

Parameters

None – This command is sent with NULL parameter, the event which is received after this command ID will contain the bit rate for the last transmitted packet.

Restrictions

This command has to be used after a TX operation is performed.

4.8.51 AP_CONFIG_COMMIT

Synopsis

This command is used to start the SoftAP mode with the given credentials.

Command Parameters: WMI_AP_CONFIG_COMMIT_CMD

Type	Name	Comment	
A_UINT8	networktype	NETWORK_TYPE	
		0x01	INFRA_NETWORK
		0x02	ADHOC_NETWORK
		0x04	ADHOC_CREATOR
A_UINT8	dot11authmode	DOT11_AUTH_MODE	
		0x01	OPEN_AUTH
		0x02	SHARED_AUTH
A_UINT8	authmode	AUTH_MODE	
		0x01	NONE_AUTH
		0x02	WPA_AUTH
		0x04	WPA2_AUTH
		0x08	WPA_PSK_AUTH
		0x10	WPA2_PSK_AUTH
		0x20	WPA_AUTH_CCKM
		0x40	WPA2_AUTH_CCKM
A_UINT8	pairwiseCryptoType	CRYPTO_TYPE	
		0x01	NONE_CRYPT
		0x02	WEP_CRYPT
		0x04	TKIP_CRYPT
		0x08	AES_CRYPT
A_UINT8	pairwiseCryptoLen	Length in bytes. Valid when the type is WEP_CRYPT, otherwise this should be 0	
A_UINT8	groupCryptoType	CRYPTO_TYPE	
A_UINT8	groupCryptoLen	Length in bytes. Valid when the type is WEP_CRYPT, otherwise this should be 0	
A_UINT8	ssidLength	SSID length for the AP mode	

Type	Name	Comment		
A_UCHAR	ssid[32]	SSID value for the SoftAP mode		
A_UINT16	channel	Channel in which the AP mode has to be started		
A_UINT8	bssid[6]			
A_UINT8	ctrl_flags	WMI_CONNECT_CTRL_FLAGS_BITS		
		CONNECT_ASSOC _POLICY_USER	0x0001	Associative frames are sent using the policy specified by the CONNECT_SEND-_REASSOC flag
		CONNECT_IGNORE _WPAx-_GROUP_CIPHER	0x0004	Ignore the WPAx group cipher for WPA/WPA2
		CONNECT_DO_WPA_OFFLOA D	0x0040	Use the authenticator in the QCA4002
		CONNECT_WPS_FLAG	0x0100	Set to indicate that the AP will add WPS IE to its beacon
			0xFFFF	Reset all control flags

Reset Value

None defined

Restrictions

None

4.8.52 AP_HIDDEN_SSID

Synopsis

This command is used to set the SSID to hidden mode in Soft AP functionality

Command Parameters: WMI_AP_HIDDEN_SSID_CMD

Type	Name	Comment
A_UINT8	hidden_ssid	A flag which will indicate whether hidden mode is to enabled or not

Reset Values

None

Restrictions

None

4.8.53 AP_INACTIVITY_TIME

Synopsis

This command is used to set the inactivity timeout value in Soft AP functionality

Command Parameters: WMI_AP_CONN_INACT_CMDID

Type	Name	Comment
A_UINT32	Period	The timeout period in seconds within which if there is no activity AP will disconnect the station

Reset Values

None

Restrictions

None

4.8.54 AP_SET_DTIM_INTERVAL

Synopsis

This command is used to set the DTIM interval in Soft AP functionality

Command Parameters: WMI_AP_SET_DTIM_CMDID

Type	Name	Comment
A_UINT8	Dtim	The DTIM interval for SoftAP mode is set in this parameter, the connected station will wake up after this interval to check for multicast packets in the AP, the DTIM interval is in terms of beacon intervals. E.g.: a DTIM value of 1 means station will wake up after 1 beacon interval and check for multicast packets

Reset Values

None

Restrictions

None

4.8.55 AP_SET_COUNTRY_CODE

Synopsis

This command is used to set the country code in Soft AP functionality

Command Parameters: WMI_AP_SET_COUNTRY_CMDID

Type	Name	Comment
A_UCHAR	countryCode[3]	The country code will be set in SoftAP, based on this the AP mode regulatory features will be enabled for that particular country code. The following country code are supported: "AR", "AM", "AT", "AU", "AZ", "BH", "BE", "BG", "BN", "BO", "BR", "BY", "BZ", "CH", "CL", "CN", "CY", "CZ", “DB", "DE", "DK", "DO", "EE", "EG", "ES", "FI", "FR", "GB", "GE", "GR", "GT", "HN", "HK", "HR", "HU", "ID", "IE", "IL", "IN", "IR", "IS", "IT", "KP", "KR", "KW", "KZ", "LI", "LT", "LU", "LV", "MA", "MC", "MK", "MY","NL", "NZ", "OM", "PA", "PE", "PH", "PK", "PL", "PT", "QA", "RO", "RU", "SA", "SE", "SG", "SK", "SI", "SV", "SY", "TH", "TT", "TN", "TW", "UA", "US", "UY", "UZ", "VE","YE", "ZA”

Reset Values

None

Restrictions

None

4.8.56 AP_PSBUFF_OFFLOAD

Synopsis

This command is used to enable power save buffering and set number of buffers when handling power save clients in Soft AP mode.

Command Parameters: WMI_AP_PSBUF_OFFLOAD_CMDID

Type	Name	Comment	
A_UINT8	Enable	0	Enable buffering mechanism in firmware to handle power save clients
		1	Disable buffering mechanism

Type	Name	Comment
A_UINT8	psBufCount	Range: 1-3 Specifies number of buffers allowed to buffer power save packets.

Reset Values

None

Restrictions

Command used only in Soft AP mode.

4.9 WMI events

Command Name	Value	Description
4.9.1 ADDBA_REQ	0x1020	Add block ACK
BSSINFO	0x1004	Contains information describing BSSs
4.9.3 CMDERROR	0x1005	QCA4002 device encountered an error while attempting to process a command
4.9.4 CONNECT	0x1002	The device has connected to a wireless network
4.9.5 DISCONNECT	0x1003	The device lost connectivity with a wireless network page
4.9.6 ERROR_REPORT	0x100D	An error has occurred for which the host previously requested notification
4.9.24 GET_BITRATE	0xF001	Get the last transmitted bit rate from the chip
4.9.23 PEER_NODE_EVENT	0x101B	The chip will send this event upon WPA 4-way handshake success
4.9.7 P2P_GO_NEG_RESULT	0x1036	Result of P2P GO Negotiation
4.9.8 P2P_INVITE_REQ	0x103E	P2P Invitation request from other P2P Device
4.9.9 P2P_INVITE_RCVD_RESULT	0x103F	Acknowledgement from P2P GO on receiving P2P invitation response.
4.9.10 P2P_PROV_DISC_RESP	0x1041	Provisional discovery response from P2P Device

Command Name	Value	Description
4.9.11 P2P_PROV_DISC_REQ	0x1042	Provisional discovery request from another P2P Dev
4.9.12 P2P_NODE_LIST	0x901A	P2P Device list from firmware
4.9.13 P2P_REQ_TO_AUTH	0x901B	P2P Connect request event from another P2P Device
4.9.14 P2P_LIST_PERSISTENT_NETWORK	0x902D	Gives an array of P2P persistent network information to the host
4.9.15 READY	0x1001	The QCA4002 device is ready to accept commands
4.9.16 REGDOMAIN	0x1006	The regulatory domain has changed
4.9.17 SCAN_COMPLETE	0x100A	A scan has completed
4.9.18 SOCKET_RESPONSE	0x9016	Returns data/status on the IP stack
4.9.20 STORERECALL_STORE	0x9004	Returns system snapshot to host
4.9.19 TKIP_MICERR	0x1009	TKIP MIC errors were detected

Command Name	Value	Description
4.9.21 WPS_GET_STATUS	0x9005	Returns WPS status/error codes
4.9.22 WPS_PROFILE	0x9006	Returns valid profile to host

4.9.1 ADDBA_REQ

Synopsis

Requests adding a block ACK.

Event ID

0x1020

Event Parameters: WMI_ADDBA_REQ_EVENT

Type	Name	Comment
A_UINT8	tid	TID sending the request
A_UINT8	win_sz	Size of the window
A_UINT16	st_seq_no	Starting sequence number
A_UINT8	status	OK or Failure

4.9.2 BSSINFO

Synopsis

Contains information describing one or more BSSs as collected during a scan. Information includes the BSSID, SSID, RSSI, network type, channel, supported rates, and IEs. `BSSINFO` events are sent only after the device receives a beacon or probe response frame that pass the filter specified in the SET_BSS_FILTER command. `BSSINFO` events consist of a small header followed by a copy of the beacon or probe response frame. The 802.11 header is not present. For formats of beacon and probe response frames please consult the IEEE 802.11 specification.

The beacons or probe responses containing the IE specified by the WMI_BSS_FILTER_CMD are passed to the host through the WMI_BSSINFO_EVENT. The event carries a 32-bit bitmask that indicates the IEs that were detected in the management frame. The frame type field has been extended to indicate action management frames. This would be helpful to route these frames through the same event mechanism as used by the beacon processing function.

If the bssFilter in the SET_BSS_FILTER matches, then the ieMask is not relevant because the

BSSINFO event is sent to the host. If the bssFilter does not match in the beacons/probe responses, then the ieMask match dictates whether the BSSINFO event is sent to the host. In the case of action management frames, the ieMask is the filter that is applied.

NOTE: Driver implementation calculates RSSI and reports event in theWMI_BS_INFO_HDR format.

Event ID

0x1004

Event Parameters: WMI_BSS_INFO_HDR

Type	Name	Comment		
A_UINT16	channel	Specifies the frequency (in MHz) where the frame was received		
A_UINT8	frameType	0x1	BEACON_FTYPE	Indicates a beacon frame
		0x2	PROBERESP_FTYPE	Indicates a probe response frame
		0x3	ACTION_MGMT_FTYPE	
		0x4	PROBEREQ_FTYPE	
A_UINT8	snr			
A_UINT8	bssid[6]			
A_UINT32	ieMask			

4.9.3 CMDERROR

Synopsis

Indicates that the QCA4002 device encountered an error while attempting to process a command. This error is fatal and indicates that the device requires a reset.

Event ID

0x1005

Event Parameters: WMI_CMD_ERROR_EVENT

Type	Name	Comment
A_UINT16	commandId	Corresponds to the command that generated the error

Type	Name	Comment		
A_UINT8	errorCode	A WMI_ERROR_CODE value; all other values reserved		
		1	INVALID_PARAM	Invalid parameter
		2	ILLEGAL_STATE	Illegal state
		3	INTERNAL_ERROR	Internal error
		4	STACK_ERROR	Stack error

4.9.4 CONNECT

Synopsis

Signals that the QCA4002 is connected to a wireless network. Connection occurs due to a connect command or roaming to a new AP. For infrastructure networks, it shows that the target successfully performed 802.11 authentication and AP association.

Event ID

0x1002

Event Parameters: WMI_CONNECT_EVENT

Type	Name	Comment
A_UINT16	channel	Specifies the frequency (in MHz) where the frame was received
A_UINT8	bssid[6]	MAC address of the AP the QCA4002 is connected to or the BSSID of the ad hoc network
A_UINT16	listenInterval	Listen interval (in Kμs) that the QCA4002 is using
A_UINT16	beaconInterval	Beacon interval the connected AP is using

A_UINT32	networkType	Network type in NETWORK_TYPE
		INFRA_NETWORK = 0x01 Infrastructure network
		ADHOC_NETWORK = 0x02 Ad hoc network
		ADHOC_CREATOR = 0x04 Ad hoc network created by the QCA4002

A_UINT8	beaconIeLen	Length (in bytes) of the beacon IEs
A_UINT8	assocReqLen	Length (in bytes) of the assocReqIEs array
A_UINT8	assocRespLen	Length (in bytes) of the assocRespIEs array

Type	Name	Comment
A_UINT8	assocInfo[]	Array containing beacon IEs, followed first by association request IEs then by association response IEs

4.9.5 DISCONNECT

Synopsis

Signals that the QCA4002 device lost connectivity with the wireless network. DISCONENCT is generated when the device fails to complete a CONNECT command or as a result of a transition from a connected state to disconnected state.

After sending the DISCONNECT event the device continually tries to re-establish a connection, if roaming is enabled. A LOST_LINK occurs when STA cannot receive beacons within the specified time for the SET_BMISS_TIME command.

Event ID

0x1003

Event Parameters: WMI_DISCONNECT_EVENT

Type	Name	Comment		
A_UINT16	Protocol ReasonStatus	Reason code (see the 802.11 specification)		
A_UINT8	bssid[6]	Indicates which BSS the device was connected to		
A_UINT16	disconnect Reason	0x01	NO_NETWORK_AVAIL	The device was unable to establish or find the desired network
		0x02	LOST_LINK	The devices is no longer receiving beacons from the BSS
		0x03	DISCONNECT_CMD	A DISCONNECT command was processed
		0x04	BSS_DISCONNECTED	Indicates the BSS explicitly disconnected the device. Possible mechanisms include the AP sending 802.11 management frames (e.g., disassociate or deauthentication messages).
		0x05	AUTH_FAILED	Indicates that the device failed 802.11 authentication with the BSS
		0x06	ASSOC_FAILED	Indicates that the device failed 802.11 association with the BSS

Type	Name	Comment		
		0x07	NO_RESOURCES_AVAIL	Indicates that a connection failed because the AP had insufficient resources to complete the connection
		0x08	CSERV_DISCONNECT	Indicates that the device's connection services module decided to disconnect from a BSS, which can happen for a variety of reasons (e.g., the host marks the current connected AP as a bad AP).
		0x0A	INVALID_PROFILE	Indicates that an attempt was made to reconnect to a BSS that no longer matches the current profile. This reason code will also be sent in case of a WPA handshake failure .
		0x0B	DOT11H_CHANNEL_SWITCH	Indicates that the AP the device is connected to has sent the DOT11H CSA IE.
		0x0C	PROFILE_MISMATCH	Indicates that the AP the device is associated to has changed its profile and the STA has detected the profile change.
		0x0E	IBSS_MERGE	Indicates the STA merged with another IBSS network
		All other values are reserved		
A_UINT16	assocRespLen	Length of the 802.11 association response frame that triggered this event, or 0 if not applicable		
A_UINT32	assocInfo [assocRespLen]	Copy of the 802.11 association response frame		

4.9.6 ERROR_REPORT

Synopsis

Signals that a type of error has occurred for which the host previously requested notification through the TARGET_ERROR_REPORT_BITM

Event ID

0x100D

Event Parameters: WMI_TARGET_ERROR_REPORT_EVENT

Type	Name	Comment	
A_UINT32	errorVal	See TARGET_ERROR_BITMASK	
		0x00000001	Power save fails
		0x00000002	No cipher key
		0x00000004	Decryption error
		0x00000008	Beacon miss
		0x00000010	A non-power save disabled node has joined the PS-enabled network
		0x00000020	Com error
		0x00000040	Fatal error
		0x00000080	Beacon found

4.9.7 P2P_GO_NEG_RESULT

Synopsis

Occurs as a results of a host-issued P2P_CONNECT command; it indicates success or failure.

Event ID

0x1036

Event Parameters: WMI_P2P_GO_NEG_RESULT_EVENT

Type	Name	Comment
A_UINT16	Freq	P2P Device operating freq if we act as P2PGO else operating freq of Peer P2P GO.
A_INT8	Status	Indicate Negotiation success or failure (-1 indicates failure and 0 indicate success)
A_UINT8	Role_go	Role_go is set to 1 if this P2P Device act asP2P GO
A_UINT8	Ssid[WMI_MAX_SSID_LEN]	Random generated SSID if P2P negotiation result into GO Mode
A_UINT8	Ssid_len	SSID length

Type	Name	Comment	
A_CHAR	Pass_phrase[WMI_MAX_PASSPHRASE_LEN]	Random generated passphrase if P2P negotiation result into GO Mode	
A_UINT8	peer_device_addr[ATH_MAC_LEN]	Peer P2P Device address	
A_UINT8	peer_interface_addr[ATH_MAC_LEN]	Peer interface MAC address	
A_UINT8	wps_method	0	WPS_NOT_READY
		1	WPS_PIN_LABEL
		2	WPS_PIN_DISPLAY
		3	WPS_PIN_KEYPAD
		4	WPS_PBC
A_UINT8	persistent_grp	Flag set to '1' if the connecting peer support P2P Persistent	

4.9.8 P2P_INVITE_REQ

Synopsis

Indicates that an invitation was received from another P2P device to join to a P2P GO; the P2P invitation is also received to invoke a persistent P2P connection.

Event ID

0x103E

Event Parameters: WMI_P2P_FW_INVITE_REQ_EVENT

Type	Name	Comment
A_UINT8	sa[ATH_MAC_LEN]	Source MAC address of P2P Device requesting invitation request
A_INT8	bssid[ATH_MAC_LEN]	P2P GO BSSID
A_UINT8	go_dev_addr[ATH_MAC_LEN]	P2P Device GO P2P Device address
P2P_SSID	ssid	SSID and SSID Length of request P2P Device
A_UINT8	is_persistent	Invitation request is also to indicate to revoke persistent connection. This flag is set for persistent connection support
A_CHAR	dialog_token	Dialog token of P2P Invitation request

4.9.9 P2P_INVITE_RCVD_RESULT

Synopsis

Indicates that an invitation was received after another P2P device sent a P2P INVITE_REQ_RSP command; once this event is received, the P2P device will join the P2P GO it was invited to.

Event ID

0x103F

Event Parameters: WMI_P2P_INVITE_RCVD_RESULT_EVENT

Type	Name	Comment	
A_UINT16	oper_freq	Operating freq of the invited P2P Device	
A_INT8	sa[ATH_MAC_LEN]	Source MAC address of P2P Device requesting invitation req	
A_UINT8	bssid[ATH_MAC_LEN]	P2P GO BSSID	
A_UINT8	is_bssid_valid	TRUE if P2P GO is running	
A_UINT8	go_dev_addr[ATH_MAC_LEN]	P2P Device GO　P2P Device address	
P2P_SSID	ssid	SSID and SSID Length	
A_UINT8	status	0	P2P_SC_SUCCESS
		1	P2P_SC_FAIL_INFO_CURRENTLY_UNAVAILABLE
		2	P2P_SC_FAIL_INCOMPATIBLE_PARAMS
		3	P2P_SC_FAIL_LIMIT_REACHED
		4	P2P_SC_FAIL_INVALID_PARAMS
		5	P2P_SC_FAIL_UNABLE_TO_ACCOMMODATE
		6	P2P_SC_FAIL_PREV_PROTOCOL_ERROR
		7	P2P_SC_FAIL_NO_COMMON_CHANNELS
		8	P2P_SC_FAIL_UNKNOWN_GROUP
		9	P2P_SC_FAIL_BOTH_GO_INTENT_15
		10	P2P_SC_FAIL_INCOMPATIBLE_PROV_METHOD
		11	P2P_SC_FAIL_REJECTED_BY_USER

4.9.10 P2P_PROV_DISC_RESP

Synopsis

A provisional discovery response that occurs when host issues a P2P_FW_PROV_DISC_REQ.

Event ID

0x1041

Event Parameters: WMI_P2P_PROV_DISC_RESP_EVENT

Type	Name	Comment	
A_UINT8	peer[ATH_MAC_LEN]	Peer MAC address responding provisional response	
A_INT16	config_methods	0x0100	WPS_CONFIG_KEYPAD
		0x0080	WPS_CONFIG_PUSHBUTTON
		0x0008	WPS_CONFIG_DISPLAY

4.9.11 P2P_PROV_DISC_REQ

Synopsis

Indicates that a P2P provisional discovery request event (to know this device's WPS supported methods) has been received from another P2P device.

Event ID

0x1042

Event Parameters: WMI_P2P_PROV_DISC_REQ_EVENT

Type	Name	Comment

Type	Name	Comment	
A_UINT8	sa[ATH_MAC_LEN]	Source MAC address of P2P Device requesting provisional disc request	
A_INT16	wps_config_method	0x0008	WPS_CONFIG_DISPLAY
		0x0080	WPS_CONFIG_PUSHBUTTON
		0x0100	WPS_CONFIG_KEYPAD
A_UINT8	dev_addr[ATH_MAC_LEN]	P2P Device MAC address of request P2P Device	
A_UINT8	pri_dev_type[WPS_DEV_TYPE_LEN]	Request P2P Device Primary device type	
		1	WPS_DEV_COMPUTER
		2	WPS_DEV_INPUT
		3	WPS_DEV_PRINTER
		4	WPS_DEV_CAMERA
		5	WPS_DEV_STORAGE
		6	WPS_DEV_NETWORK_INFRA
		7	WPS_DEV_DISPLAY
		8	WPS_DEV_MULTIMEDIA
		9	WPS_DEV_GAMING
		10	WPS_DEV_PHONE
A_UINT8	device_name[WPS_MAX_DEVNAME_LEN]	Device name of requesting P2P Device	
A_UINT8	dev_name_len	Device name length	
A_UINT16	dev_config_methods	P2P Device supported configuration methods	
A_UINT8	device_capab	P2P_DEV_CAPAB_SERVICE_DISCOVERY BIT(0) P2P_DEV_CAPAB_CLIENT_DISCOVERABILITY BIT(1) P2P_DEV_CAPAB_CONCURRENT_OPER BIT(2) P2P_DEV_CAPAB_INFRA_MANAGED BIT(3) P2P_DEV_CAPAB_DEVICE_LIMIT BIT(4) P2P_DEV_CAPAB_INVITATION_PROCEDURE BIT(5)	

Type	Name	Comment
A_UINT8	group_capab	P2P_GROUP_CAPAB_GROUP_OWNER BIT(0) P2P_GROUP_CAPAB_PERSISTENT_GROUP BIT(1) P2P_GROUP_CAPAB_GROUP_LIMIT BIT(2) P2P_GROUP_CAPAB_INTRA_BSS_DIST BIT(3) P2P_GROUP_CAPAB_CROSS_CONN BIT(4) P2P_GROUP_CAPAB_PERSISTENT_RECONN BIT(5) P2P_GROUP_CAPAB_GROUP_FORMATION BIT(6)

4.9.12 P2P_NODE_LIST

Synopsis

A P2P nodelist event that occurs when the host issues P2P_GET_NODE_LIST. This event is clubbed with between information from one and three P2P devices. Note that firmware can populate up to five P2P devices and the next immediate P2P_GET_NODE_LIST command will fetch remaining P2P devices.

Event ID

0x901A

Event Parameters: WMI_P2P_NODE_LIST_EVENT

Type	Name	Comment
A_UINT8	num_p2p_dev	Number of P2P Device populated in the event

Type	Name	Comment

A_INT8	data[1]	Num_p2p_dev * (struct p2p_device)
		Struct p2p_device {
		A_UINT16 listen_freq;
		A_UINT32 wps_pbc;
		A_CHAR wps_pin[WPS_MAX_PIN_LEN];
		A_UINT8 pin_len;
		A_UINT32 wps_method;
		A_UINT8 p2p_device_addr[ETH_ALEN];
		A_UINT8 pri_dev_type[8];
		A_CHAR device_name[33];
		A_UINT16 config_methods;
		A_UINT8 dev_capab;
		A_UINT8 group_capab;
		A_UINT8 interface_addr[ETH_ALEN];
		A_UINT8 dev_disc_dialog_token;
		A_UINT8 member_in_go_dev[ETH_ALEN];
		A_UINT8 member_in_go_iface[ETH_ALEN];
		A_UINT8 dev_disc_go_oper_ssid[WMI_MAX_SSID_LEN];
		A_UINT8 dev_disc_go_oper_ssid_len;
		A_UINT16 dev_disc_go_oper_freq;
		A_UINT32 go_neg_req_sent;
		A_UINT32 go_state;
		A_UINT8 dialog_token;
		A_UINT8 intended_addr[ETH_ALEN];
		A_CHAR country[3];
		struct p2p_channels channels;
		A_UINT16 oper_freq;
		A_UINT8 oper_ssid[P2P_MAX_SSID_LEN];
		A_UINT8 oper_ssid_len;
		A_UINT16 req_config_methods;
		A_UINT32 flags;
		A_UINT32 status;
		A_UINT16 ext_listen_interval;
		A_UINT8 go_timeout;
		A_UINT8 client_timeout;
		A_UINT8 persistent_grp; };
		A_UINT16 ext_listen_period;
		A_UINT32 invitation_reqs;
		A_UINT8 wait_count;

4.9.13 P2P_REQ_TO_AUTH

Synopsis

Received from another P2P device that wishes to connect to this P2P device.

Event ID

0x901B

Event Parameters: WMI_P2P_REQ_TO_AUTH_EVENT

Type	Name	Comment	
A_UINT8	sa[ATH_MAC_LEN]	Source MAC address of P2P Device requested connection.	
A_INT8	Dialog_token	Dialog token of P2P AUTH Request	
A_UINT16	Dev_password_id	0x0000	DEV_PW_DEFAULT
		0x0001	DEV_PW_USER_SPECIFIED
		0x0002	DEV_PW_MACHINE_SPECIFIED
		0x0003	DEV_PW_REKEY
		0x0004	DEV_PW_PUSHBUTTON
		0x0005	DEV_PW_REGISTRAR_SPECIFIED

4.9.14 P2P_LIST_PERSISTENT_NETWORK

Synopsis

Event occurs for P2P_LIST_PERSISTENT_CMD; it gives an array of P2P persistent network information to the host. By default the event returns 5 persistent network information.

Event ID

0x902D

Event Parameters: WMI_P2P_REQ_TO_AUTH_EVENT

Type	Name	Comment

Type	Name	Comment

A_UINT8	Data[1]	Persistent information
		{
		A_UINT8 role;
		A_UINT8 macaddress[6];
		A_UINT8 ssid[32];
		}

4.9.15 READY

Synopsis

Indicates that the target device is prepared to accept commands. This even is sent once after power on or reset. It also indicates the MAC address of the device.

Event ID

0x1001

Event Parameters: WMI_READY_EVENT_2

Type	Name	Comment	
A_UINT32	sw_version		
A_UINT32	abi_version	Binary version of the firmware Used by the driver to ensure the firmware is compatible with the driver	
A_UINT8	macAddr[ATH_MAC_LEN]	Device MAC address	
A_UINT8	phyCapability	Indicates the capabilities of the device WMI radio; all other values reserved	
		1	WMI_11A_CAPABILITY
		2	WMI_11G_CAPABILITY
		3	WMI_11AG_CAPABILITY
		4	WMI_11NA_CAPABILITY
		5	WMI_11NG_CAPABILITY
		6	WMI_11NAG_CAPABILITY

4.9.16 REGDOMAIN

Synopsis

Indicates that the regulatory domain has changed. It initially occurs when the QCA4002 device reads the board data information. The regulatory domain can also change when the device is a world-mode SKU. In this case, the regulatory domain is based on the country advertised by APs per the IEEE 802.11d specification. A potential side effect of a regulatory domain change is a change in the list of available channels. Any channel restrictions that exist as a result of a previous “” are lifted.

Event ID

0x1006

Event Parameters: WMI_REG_DOMAIN_EVENT

Type	Name	Comment
A_UINT32	regDomain	The range of 0x0000– 0x00FF corresponds to an ISO country code. Other regCodes are reserved for world mode settings and specific regulatory domains.

4.9.17 SCAN_COMPLETE

Synopsis

Indicates the scan status. if the Scan was not completed, this event is generated with the status A_ECANCELED. If the scan cannot start, this event is generated with status A_EBUSY. If the scan completes successfully, this event is generated with status A_OK.

Event ID

0x100A

Event Parameters: WMI_SCAN_COMPLETE_EVENT

Type	Name	Comment		
A_INT32	Status	A WMI_ERROR_CODE value; all other values reserved		
		0	A_OK	Success
		13	A_EBUSY	Error: Scan busy

Type	Name	Comment		
		16	A_ECANCELLED	Error: Cancelled

4.9.18 SOCKET_RESPONSE

Synopsis

Socket events are used to communicate status on Socket Events from the target QCA4002 to the host. This is in response to a Socket command.

NOTE: This event is only present in the QCA4002.

Event ID

0x9016

Event Parameters

Type	Name	Comment		
A_UINT32	resp_type	Socket command response type		
		1	SOCK_STREAM_TYPE	TCP
		2	SOCK_DGRAM_TYPE	UDP
A_UINT32	sock_handle	Socket handle		
A_UINT32	error	Return value. Supported values are:		
		-1	Indicates the command Failed	
		0	Indicates the command passed	
A_UINT8	data[1]	Data for socket command; commands responses can be for the following commands:		
		SOCK_OPEN	Open a socket	
		SOCK_CLOSE	Close existing socket	
		SOCK_CONNECT	Connect to a peer	
		SOCK_BIND	Bind to interface	
		SOCK_LISTEN	Listen on socket	
		SOCK_PING	Response of outcome of an IPv4 ping	
		SOCK_PING6	Response of outcome of an IPv6 ping	

		SOCK_ACCEPT	Accept incoming connection:
			} SOCK_CONNECT_T, SOCK_BIND_T, SOCK_ACCEPT_T;
			typedef struct sock_connect {
			A_UINT32 handle; //socket handle
			union
			{
			SOCKADDR_T name;
			SOCKADDR_6_T name6;
			} addr; // IP Address and port
			A_UINT16 length; //socket address length	

Type	Name	Comment		

		SOCK_IPCONFIG	Get socket option:
			typedef struct sock_setopt
			{
			A_UINT32 handle; //socket handle
			A_UINT32 level;
			// Option level (ATH_IPPROTO_IP,
			// ATH_IPPROTO_UDP, ATH_IPPROTO_TCP...)
			A_UINT32 optname; //Choose from list above
			A_UINT32 optlen; // option length
			A_UINT8 optval[1]; // option value
			} SOCK_OPT_T;	

		SOCK_IP6CONFIG	Set static IP information, or get current IPv4 configuration	typedef struct ipconfig {
				A_UINT32 mode;
				// IPCFG_QUERY=0, Retrieves the IP parameters
				// IPCFG_STATIC=1, Set static IP parameters
				// IPCFG_DHCP=2, Run DHCP client
				A_UINT32 ipv4; // IPv4 address
				A_UINT32 subnetMask; // IPv4 subnet mask
				A_UINT32 gateway4; // IPv4 gateway
				IP6_ADDR_T ipv6LinkAddr; // IPv6 link local addr
				IP6_ADDR_T ipv6GlobalAddr; // IPv6 global addr
				IP6_ADDR_T ipv6DefGw; // IPv6 Default Gateway
				IP6_ADDR_T ipv6LinkAddrExtd; // IPv6 link lcl addr
				A_INT32 LinkPrefix;
				A_INT32 GlbPrefix;
				A_INT32 DefGwPrefix;
				A_INT32 GlbPrefixExtd;
				} IPCONFIG_T;

		SOCK_IP6CONFIG	Set static IP information, or get current IPv6 configuration	

		SOCK_DNC_COMMAND	The command to resolve host name
			typedef PREPACK struct {
			A_UINT32 resp_type;
			A_UINT32 handle;
			A_UINT32 result;
			DNC_RESP_INFO data
			}POSTPACK
			WMI_DNC_RESPONSE_EVENT
			
			typedef PREPACK struct
			dncrespinfo {
			A_UINT8 dns_names [256]
			FIELD_PACKED;
			A_INT32 h_length FIELD_PACKED;
			A_INT32 h_addrtype
			FIELD_PACKED;
			A_INT32 ipaddrs FIELD_PACKED;
			A_UINT32
			ipaddrs_list[MAX_DNSADDRS]
			FIELD_PACKED;
			A_INT32 ip6addrs FIELD_PACKED;
			IP6_ADDR_T
			ip6addrs_lis[MAX_DNSADDRS]
			FIELD_PACKED;
			} POSTPACK DNC_RESP_INFO;
			
			The above structure fills the relevant information extracted from the answer received from the server for a DNS Client Query

4.9.19 TKIP_MICERR

Synopsis

Indicates that TKIP MIC errors were detected.

Event ID

0x1009

Event Parameters: WMI_TKIP_MICERR_EVENT

Type	Name	Comment	
A_INT8	Keyed	Indicates the TKIP key ID	
A_INT8	ismcast	0	Unicast
		1	Multicast

4.9.20 STORERECALL_STORE

Synopsis

Signals that the AR4001 device ready to sleep and send the snapshot of system data to the host.

Event ID

0x9004

Event Parameters

Type	Name	Comment
A_UINT32	msec_sleep	Time between power off
A_UINT32	length	WPS Error Codes
A_UINT8	data[1]	Starting pointer of data payload

4.9.21 WPS_GET_STATUS

Synopsis

Indicates the WPS 8-way handshake message status.

Event ID

0x9005

Event Parameters

Type	Name	Comment
A_UINT8	Status	WPS Status Codes
A_UINT8	error_code	WPS Error Codes

4.9.22 WPS_PROFILE

Synopsis

Indicates that the WPS PROFILE has been successfully retrieved from the AP.

Event ID

0x9006

Event Parameters

Type	Name	Comment

Type	Name	Comment
A_UINT8	Status	WPS Status Codes
A_UINT8	error_code	WPS Error Codes
WPS_CREDENTIAL	Credential	Retrieved credential from AP

4.9.23 PEER_NODE_EVENT

Synopsis

Indicates that WPA 4-way handshake is successful.

Event ID

0x101B

Event Parameters

Type	Name	Comment
A_UINT8	Success code	The status code is PEER_NODE_JOIN_EVENT = 0x00

4.9.24 GET_BITRATE

Synopsis

This event will get the last transmitted bit rate from QCA4002 chip.

Event ID

0xf001

Event Parameters

Type	Name	Comment
A_UINT8	rateIndex	The rateIndex indicated the rate at which the data is transmitted; there will be a table in the driver which will map the rateIndex to its appropriate value in kbps.

Note that for the GET_BITRATE command, the command ID and event ID are the same, in fact in the firmware the QCA4002 uses the same command ID.

AT Commands

The QCA4002 supports hostless UART mode using a simple AT command set. In this mode, the QCA4002 can be used as a Wi-Fi serial modem, so that two devices can have point-to-point communication over WLAN using the UART interface without requiring any driver software in these devices.

This mode requires different firmware and the QCA4002 bootstrap pins must be set to hostless mode. In hostless UART mode, the QCA4002 uses UART1 as the AT command interface. The SPI Module pins 5 and 6 are allocated as a UART1 peripheral and act as UART_RXD and UART_TXD respectively. The command set is described in Table 错误!文档中没有指定样式的文字。-1.

Table 错误!文档中没有指定样式的文字。**-1 Available AT Commands**

Command	Description		
ATHELP=	Enter an AT command followed by parameters, for example: ATWS=*parameters*		
ATW=	Get WLAN status		
ATWREV=	Return WLAN firmware version		
ATWS=	Start WLAN scan		
ATPM= *<enable>,<mode>, <wake period>,<sleep period>*	Set power mode		
	enable	0	Enable
		1	Disable
	mode	0	One shot
		1	Periodic
	wake period	Wake time in mseconds	
	sleep period	Sleep time in mseconds	
ATWA= *<ssid>*	Associate to an SSID, no security, open mode		
	ssid	SSID to associate to	
ATWAWPS= *<1>,<PUSH/PIN>*	Associate using WPS		
	1	Enable WPS	
	PUSH/PIN	Enable PUSH/PIN	

Command	Description		
ATWAWPA= *<ssid>,<wpa ver>,<ucipher>,<mcipher>, <passphrase>*	Associate using WPA		
	ssid	SSID to associate to	
	wpa ver	1	WPA
		2	WPA2
	ucipher	0	TKIP
		1	CCMP
	mcipher	0	TKIP
		1	CCMP
	passphrase	Passphrase	
ATWD=	Dissociate from connected AP		
ATWPHYMODE=*<a/b/g/n>*	Set WLAN PHY mode to 802.11a, 802.11b, 802.11g, or 802.11n		
ATWRSSI=	Get RSSI		
ATWWEPKEY= *<key index>,<hex key>*	Set WLAN WEP key		
	key index	Key index 1 to 4	
	hex key	Hexadecimal key	
ATWAWEP= *<key index>,<ssid>*	Associate using WEP mode		
	key index	Key index 1 to 4	
	ssid	SSID to associate to	
ATNPING= *<IP addr>*	Ping to specified host		
	IP addr	Host to ping to, in a.b.c.d format	
ATNSET= ? or *<IP>,<Mask>,<GW>*	Get/set network parameters		
	?	Shows list of parameters	
	IP	IP address to get/set	
	Mask	IP mask	
	GW	IP gateway	
ATNDHCP=*<1>*	Run DHCP client		
ATNSOCK= *<TCP/UDP>*	Open socket		
	0	TCP	
	1	UDP	
ATNCLOSE=	Close socket		

Command	Description	
ATNCTCP= *<IP>*,*<PORT>*	Attempt TCP client connection to destination	
	IP	Client IP address
	PORT	Client port address
ATNCUDP= *<IP>*,*<PORT>*	Attempt UDP client connection to destination	
	IP	Client IP address
	PORT	Client port address
ATNSTCP= *<PORT>*	Attempt TCP server connection	
	PORT	Client port address
ATO=	Enter transparent mode	
ATBSIZE=*<buffer size>*	Enter the buffer size <1-1200>	
ATTO=*<ms>*	Enter the timeout in ms (must be >200)	