Atmel SmartConnect

AT14596: TCP Client and Server Operation using ATWINC1500

APPLICATION NOTE

Introduction

This application note describes the TCP client and server implementations using the Atmel® ATWINC1500 Wi-Fi® Network Controller to build state-of-the-art Internet of Things (IoT) applications.

Features

- Basic concepts of Transmission Control Protocol (TCP)
- TCP client implementation example using ATWINC1500
- TCP server implementation example using ATWINC1500

Table of Contents

1. Overview

Transmission Control Protocol (TCP) is a Transport Layer Protocol that provides a reliable end-to-end service delivering packets between applications running in devices in an IP network. These packets are delivered in sequence without any loss or duplication. TCP emphasizes reliable transmission over a faster data stream and is commonly used in applications such as FTP, e-mail, and so on.

A TCP client or server application can be developed in the host Microcontroller(MCU) to which the ATWINC1500 is connected. The ATWINC1500 implements a TCP/IP networking stack that is an IP v4.0 stack based on the uIP TCP/IP stack (pronounced micro IP).

The ATWINC1500 TCP/IP stack adds to the original uIP implementation several enhancements to boost TCP and UDP throughput.

2. TCP Basic Concepts

TCP is a connection-oriented protocol that guarantees that all bytes will be transmitted and received in the correct order. For this, it uses a scheme with acknowledgments and retransmissions to guarantee reliability of packet transfers. This fundamental technique requires the receiver to respond with an acknowledgment message when it receives the data. The sender keeps a record of each packet it sends. The sender also maintains a timer from when the packet was sent and retransmits a packet if the timer expires before the message has been acknowledged. The timer is needed in case a packet gets lost or corrupted.

2.1. Sockets

TCP establishes a full-duplex virtual connection between two endpoints. Each endpoint or socket is defined by an IP address and a TCP port number. The ATWINC1500 socket API provides a method that allows the host MCU application to interact with intranet and remote internet hosts. The API is based on BSD (Berkeley) sockets. Each ATWINC1500 socket is identified by a unique combination of:

- **Socket ID:**It is a unique identifier for each socket. This is the return value of the socket API.
- **Local socket address:**A combination of ATWINC1500 IP address and port number assigned by the ATWINC1500 firmware for the socket.
- **Protocol:**This is the transport layer protocol, either TCP or UDP.
- **Remote socket address:** Applicable only for TCP stream sockets. This is necessary since TCP is connection oriented. The remote socket address can be obtained in the socket event callback in the application.

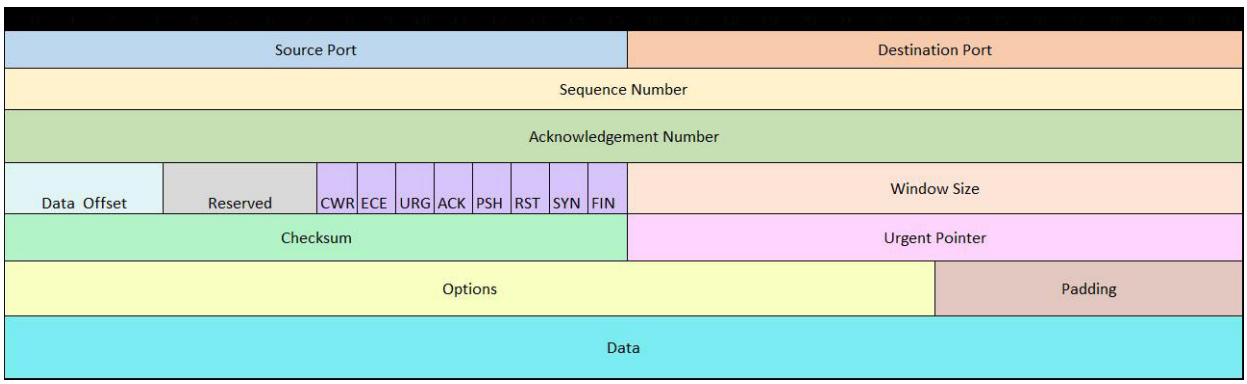
2.2. TCP Packet Structure and Size

Every TCP packet is comprised of a header and a data portion. The data portion is generally referred to as a TCP segment.

The TCP packet from the transport layer is encased in an IP packet and transmitted after the appropriate header and footer information are added in every layer of the TCP/IP stack.

The packet structure of a TCP packet is shown in the following figure.

Figure 2-1. TCP Packet Structure



2.2.1. TCP Header Flags

The TCP header defines six important flag bits. The description are as follows:

- **SYN:** for Synchronize; marks packets that are part of the new-connection handshake

- **ACK:** indicates that the Acknowledgment field is valid
- **FIN:** for Finish; marks packets involved in the connection closure
- **PSH:** for Push; marks “non-full” packets that should be delivered promptly at the destination
- **RST:** for Reset; indicates various error conditions
- **URG:** for Urgent; part of a now-seldom-used mechanism for high-priority data
- **CWR and ECE:** part of the Explicit Congestion Notification mechanism

2.2.2. TCP Packet Size

A packet is sized as per the lowest MTU (Maximum Transmission Unit) on the path, which is typically 1500 bytes (for Ethernet) of data including TCP (20 bytes) and IP headers (20 bytes). The packet size is defined in `main.h` as :

```
#define MAIN_WIFI_M2M_BUFFER_SIZE 1460
```

The application then defines a socket buffer of this size in `main.c` as:

```
static uint8_t gau8SocketTestBuffer[MAIN_WIFI_M2M_BUFFER_SIZE];
```

2.2.3. TCP Connection Establishment

TCP connections are established using an exchange known as the three-way handshake.

- A TCP client sends a listening server a packet with the SYN bit set (a SYN packet)
- The TCP server responds with a SYN packet of its own with the ACK bit also set
- The TCP client responds to the TCP server’s SYN with its own ACK

2.2.4. TCP Sliding Window Data Transfer

As TCP is connection-oriented, every TCP packet should be acknowledged before another packet can be transmitted. This reduces the throughput of the system. One method to overcome this limitation is to use a sliding window data transfer scheme which depends on the memory and processing capabilities of the sender and receiver devices. The sender might overload the receiver by sending data at a faster rate before the receiver finishes processing all incoming packets.

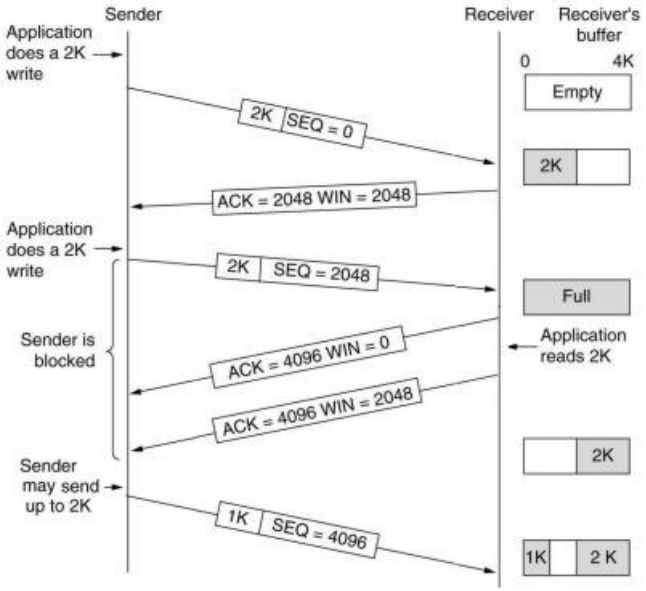
The TCP window is the amount of unacknowledged data a sender can send on a particular connection before it gets an acknowledgment back from the receiver. The receiver specifies the Window size (in bytes) in the TCP header of the ACK packet.

Flow control is managed in TCP by reducing or increasing window size, the server and client each ensure that the other device sends data just as fast as the recipient can deal with it. The "sliding" part, of sliding window, refers to the fact that the window size is negotiated dynamically during the TCP session.

2.2.5. Sequence Numbers and Acknowledgements

In TCP, the sequence numbers and acknowledgement numbers are cumulative. The receiver sends back the acknowledgment number as the number of bytes it has received successfully and the sender then proceeds to transmit data after incrementing the sequence number.

Figure 2-2. Windowing Sequence Example



TCP Zero Window - Occurs when a receiver sends an ACK with receive window size of zero. This effectively tells the sender to stop sending because the receiver's buffer is full. This is the 4th packet in the sequence above where WIN is set to zero.

TCP Window Update - A TCP Window Update occurs when the application on the receiving side has processed the data in the RX buffer and can indicate that there is now more space available in the buffer. It is typically seen after a TCP Zero Window condition has occurred. Once the application on the receiver retrieves data from the TCP buffer, thereby freeing up space, the receiver should notify the sender that the TCP Zero Window condition no longer exists by sending a TCP Window Update that advertises the current window size.

2.2.6. TCP Connection Closure

To close the TCP connection, an exchange similar to connection establishment, involving FIN packets may occur.

- The TCP client sends the TCP server a packet with the FIN bit set (a FIN packet), announcing that it has finished sending data.
- The TCP server sends the client an ACK of the FIN
- When the TCP server has processed the data and is also ready to cease sending, it sends its own FIN to the client
- The TCP client acknowledges the FIN from the server

3. TCP Client Application using ATWINC1500

This section outlines the TCP client application example from Atmel Software Framework (ASF). This application can be run on a host MCU to which the ATWINC1500 must be connected.

The TCP client can be tested with any standard TCP server implementation or with the TCP server example from ASF.

3.1. Application Overview

The TCP client application allows the user to connect to a remote TCP server using a socket and transfer data to and from the TCP server. The reference application implements transfer of a single data packet and this can be extended as per user requirements.

Other ATWINC1500 examples such as sending e-mail, locating IP Address using HTTP, and so on use TCP sockets to send and receive application data packets.

3.2. Configuration

The configuration parameters are present in `main.h` of the TCP client application. The parameters of the Access Point (AP) in the Wi-Fi® network, to which the ATWINC1500 joins as a station (STA) are:

```
#define MAIN_WLAN_SSID                    "DEMO_AP" /**< Destination SSID */
#define MAIN_WLAN_AUTH                    M2M_WIFI_SEC_WPA_PSK /**< Security manner */
#define MAIN_WLAN_PSK                     "12345678" /**< Password for Destination SSID */
#define MAIN_WIFI_M2M_PRODUCT_NAME        "NMCTemp"
```

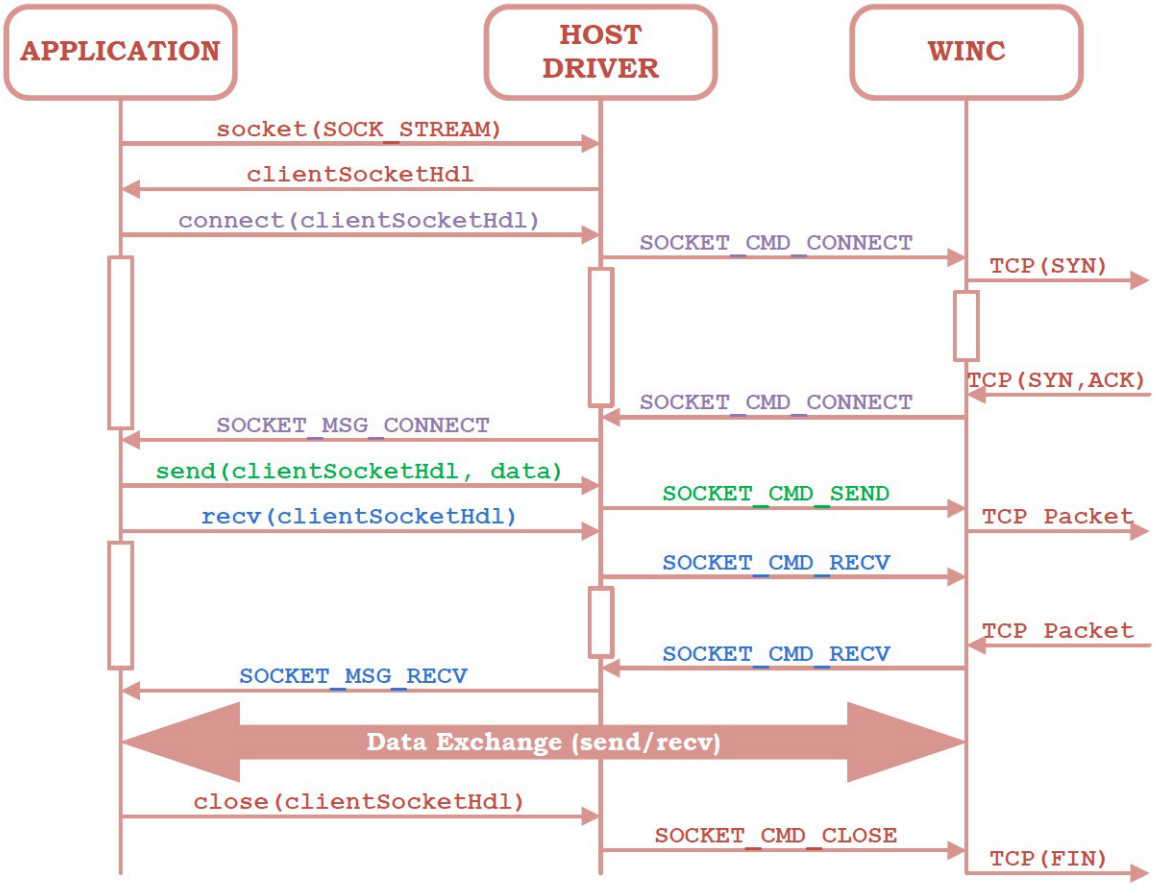
The TCP server socket related parameters are:

```
#define MAIN_WIFI_M2M_SERVER_IP           0xc0a80164 //0xFFFFFFFF /* 255.255.255.255 */
#define MAIN_WIFI_M2M_SERVER_PORT         (6666)
```

3.3. Application Structure

The overall transaction sequence for a TCP client is shown in the following diagram.

Figure 3-1. TCP Client Application Flow



3.3.1. System and Board Initialization

`system_init()`: System initialization includes host MCU initialization including clocks, external interrupts, and host MCU board.

`nm_bsp_init()`: This routine resets the ATWINC1500 through `nm_bsp_reset()` which covers the power-up sequence of the ATWINC1500.

3.3.2. Socket Setup and Initialization

A socket is setup by initializing the ATWINC1500 socket structure of type `sockaddr_in`.

```
/* Initialize socket address structure. */
addr.sin_family = AF_INET;
addr.sin_port = _htons(MAIN_WIFI_M2M_SERVER_PORT);
addr.sin_addr.s_addr = _htonl(MAIN_WIFI_M2M_SERVER_IP);
```

- `sin_family` represents the address family (AF). Currently IPv4 is supported. The IPv4 address family is defined by `AF_INET`.
- `sin_port` represents the port number of the socket. `_htons()` is used to set this 16-bit value in the network byte order format.
- `sin_addr` represents the socket address. `_htonl()` is used to set this 32-bit value in the network byte order format.

The socket is initialized using the routine from `socket.c` using the code snippet:

```
/* Initialize socket module */
socketInit();
registerSocketCallback(socket_cb, NULL);
```

As part of socket initialization, the socket buffers for the maximum number of sockets (for TCP, it is 7) are reset. Also, the callback `m2m_ip_cb()` which delivers messages to the socket component is initialized.

The application has to register a callback to receive the socket messages. This application callback will be invoked from the socket component callback `m2m_ip_cb()`.

When registering the application socket callback using `registerSocketCallback()`, there is a provision to register a callback for DNS resolution through the second argument.

3.3.3. Wi-Fi Host MCU Driver Initialization

The ATWINC1500 host MCU driver is initialized using the routine from `m2m_wifi.c`.

```
/* Initialize Wi-Fi driver with data and status callbacks. */
param.pfAppWifiCb = wifi_cb;
ret = m2m_wifi_init(&param);
if (M2M_SUCCESS != ret) {
                          printf("main: m2m_wifi_init call error!(%d)\r\n", ret);
                          while (1) {
                          }
                        }
```

`m2m_wifi_init()` initializes the bus interface (SPI) which connects the host MCU to the ATWINC1500, sets up the interrupt line from the ATWINC1500 to the host MCU and registers the Wi-Fi callback to notify the status of the wireless connection to the application.

```
static void wifi_cb(uint8_t u8MsgType, void *pvMsg)
{
    switch (u8MsgType) {
    case M2M_WIFI_RESP_CON_STATE_CHANGED:
    {
        tstrM2mWifiStateChanged *pstrWifiState = (tstrM2mWifiStateChanged *)pvMsg;
        if (pstrWifiState->u8CurrState == M2M_WIFI_CONNECTED) {
            printf("wifi_cb: M2M_WIFI_RESP_CON_STATE_CHANGED: CONNECTED\r\n");
            m2m_wifi_request_dhcp_client();
    }
       ........................
       ................................
}
```

The possible status values that could be obtained via this callback include the status of connection, status after mode setting in the ATWINC1500, DHCP and IP configuration status.

3.3.4. Connecting to an AP in a Wi-Fi network

The ATWINC1500 is now ready to connect into a Wi-Fi network. This is triggered from the host MCU using the routine from `m2m_wifi.c` as shown.

```
/* Connect to router. */
m2m_wifi_connect((char *)MAIN_WLAN_SSID, sizeof(MAIN_WLAN_SSID), MAIN_WLAN_AUTH, (char
*)MAIN_WLAN_PSK, M2M_WIFI_CH_ALL);
```

The application developer can configure the SSID, passphrase, and Wi-Fi channel number of the AP to which the ATWINC1500 will connect to,using the corresponding definitions.

3.3.5. Interrupt Handling

The Host Interface Layer (HIF) interrupts the host MCU using an external interrupt when one or more events are pending in ATWINC1500 firmware. In order to receive event callbacks, the host MCU

application needs to call the `m2m_wifi_handle_events()` to let the host driver retrieve and process the pending events from the ATWINC1500 firmware. It is recommended to call this function either:

- in the main loop of the application or in a dedicated task in the host MCU (or)
- at least once when host MCU receives an interrupt from ATWINC1500 firmware

In `main.c` of the TCP client application, the API is called in the infinite while loop.

```
while (1) {
        /* Handle pending events from network controller. */
        m2m_wifi_handle_events(NULL);
              ……
              ……
}
```

3.3.6. Connecting to a TCP Server

A TCP client socket is the logical end-point to establish a connection with the TCP server. A socket can be created by using the `socket()` API.

```
/** Socket for client */
static SOCKET tcp_client_socket = -1;

/* Open client socket. */
    if (tcp_client_socket < 0) {
      if ((tcp_client_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        printf("main: failed to create TCP client socket error!\r\n");
        continue;
 }
```

- The first argument, AF_INET indicates that only IPv4 transport addresses are supported by the ATWINC1500 firmware
- The second argument, SOCK_STREAM specifies that this is a TCP socket in contrast to the SOCK_DGRAM for UDP sockets
- The third argument specifies the flag value. It shall be left to 0 for normal TCP sockets and if a secure socket is to be created, the argument shall be populated with value SOCKET_FLAGS_SSL.

After the TCP socket is created, the socket ID (in this case, `tcp_client_socket`) is used by most other socket functions and is also passed as an argument to the socket event callback function to identify which socket generated the event.

Connection to the TCP server is initiated through the `connect()` API.

```
/* Connect server */
ret = connect(tcp_client_socket, (struct sockaddr *)&addr, sizeof(struct sockaddr_in));
```

The `connect()` API uses the socket ID (tcp_client_socket) and the address of the TCP server to connect with the remote TCP server.

3.3.7. Handling the Socket Event Callback

When the socket event callback is registered using the `registerSocketCallback()` API, socket event notifications are delivered to application through this callback. In the `main.c` of the TCP client application,

```
static void socket_cb(SOCKET sock, uint8_t u8Msg, void *pvMsg)
{
    switch (u8Msg) {
    /* Socket connected */
    case SOCKET_MSG_CONNECT:
    {
        tstrSocketConnectMsg *pstrConnect = (tstrSocketConnectMsg *)pvMsg;
              ...
       }
```

```
    }
}
```

There are several useful notifications presented through this socket callback.

Table 3-1. Event Notifications in the Socket Callback Function

msgType	Event
SOCKET_MSG_CONNECT	This notification is given to the socket call-back on attempt to connect(). The connect event will be sent when the TCP server accepts the connection or, if no remote host response is received, after a timeout interval of approximately 30 seconds. In the TCP client application, on successful connection establishment, send() is called to transmit an application data packet from client to remote server.
SOCKET_MSG_SEND	This notification is given to the socket callback after the data is transmitted to the remote TCP server. For TCP sockets, this event guarantees that the data has been delivered to the TCP server. The event callback will return the size of the data transmitted if the transmission in the success case and zero or negative value in case of an acknowledgment not being received.
SOCKET_MSG_RECV	When data reception is initiated through recv(), this event callback arrives when the received data is available in the application buffer. In the TCP client application, gau8SocketTestBuffer is allocated as the application buffer.

3.3.8. Transmitting Data to the TCP Server

Transmission of data to the TCP server is initiated through the send() API.

```
int16_t ret;
ret = send(tcp_client_socket, &msg_wifi_product, sizeof(t_msg_wifi_product), 0);
```

The send() API uses the socket ID (tcp_client_socket), the application data payload, its length, and a flag to indicate if this is a secure or unsecure transmission. The return value of send() will be zero in case of successful initiation of transmission (and will not indicate the status of the transmission itself). In case of error conditions, a negative error code is returned.

> When all the buffers in the ATWINC1500 are in use, SOCK_ERR_BUFFER_FULL(-14) is returned. In this case, the application needs to suspend data transfer for a time period and then try to send data again. There is no notification on free buffer availability at present from the ATWINC1500 stack.

3.3.9. Receiving data from the TCP Server

The host MCU application calls the recv() API with a pre-allocated buffer.

```
recv(tcp_client_socket, gau8SocketTestBuffer, sizeof(gau8SocketTestBuffer), 0);
```

The recv() API uses the socket ID (tcp_client_socket), the application buffer containing the received data, its length and a timeout value in milliseconds. If the value is set to ZERO, the timeout will

be set to infinite (the `recv` function waits forever). If the timeout period is elapsed with no data received, the socket will get a timeout error.

When the `SOCKET_MSG_RECV` event callback arrives, this buffer will contain the received data. The received data size indicates the status:

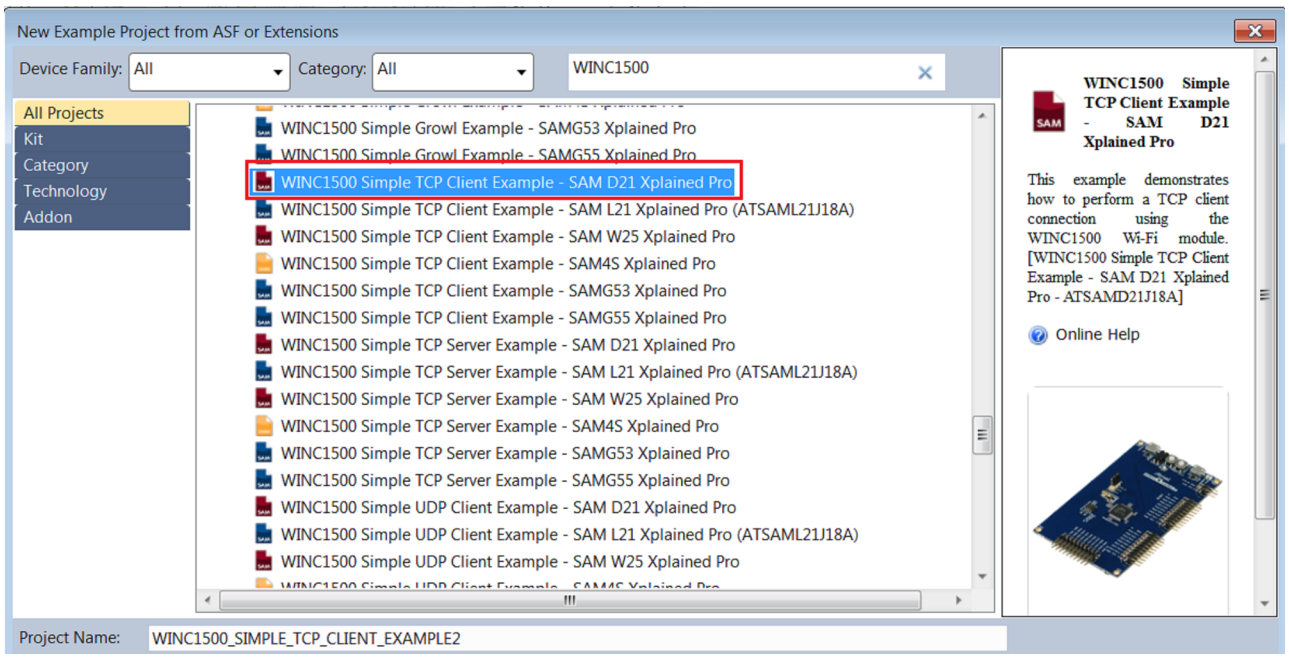
- Positive: data received
- Zero: socket connection is terminated
- Negative value: This indicates an error as explained in Transmitting Data to the TCP Server

Note: In the case of TCP sockets, it is recommended to call the `recv()` function after each successful socket connection (client or server). Otherwise, received data will be buffered in the ATWINC1500 firmware wasting the system resources until the socket is explicitly closed using a `close()` function call. The close function is used to release the resources allocated to the socket and, for a TCP stream socket, also to terminate an open connection.

3.4. Getting Started with the TCP Client Application in ASF

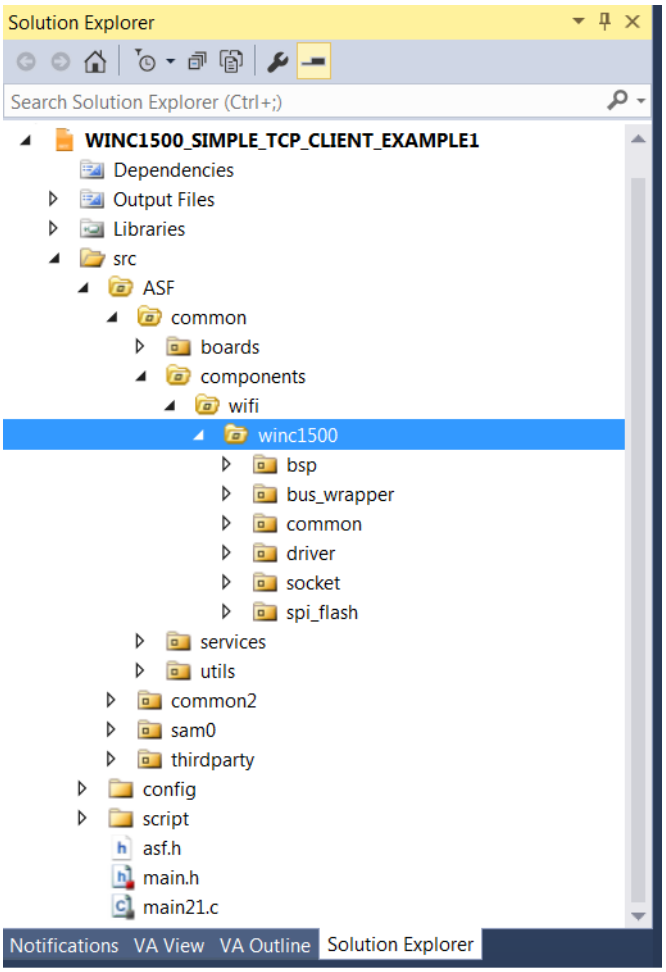
1. Open Atmel Studio 7.0 **File** > **New** > **Example Projects**.
2. Search for WINC1500 in the Search bar on the top right corner. A list of ATWINC1500 projects is displayed.
3. Select the WINC1500 Simple TCP Client Example Application for SAM D21 as shown.

Figure 3-2. TCP Client Example Application



The directory structure for the TCP client application is shown.

Figure 3-3. Application Directory Structure

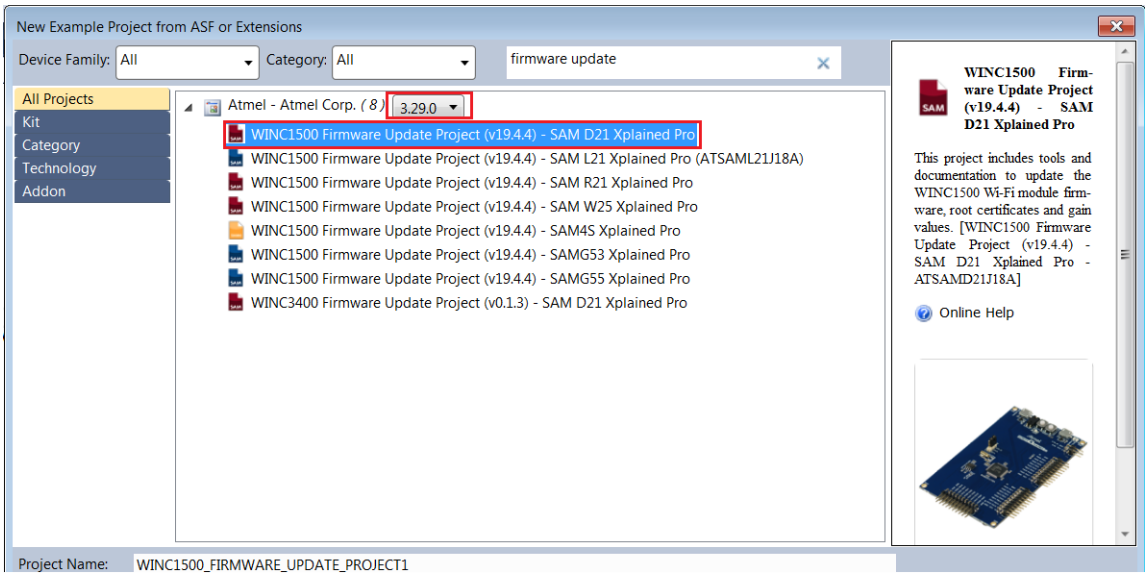


3.4.1. Programming the SAM D21 and ATWINC1500

Before programming the application on the SAM D21 Host MCU, the user must ensure that the firmware on the ATWINC1500 is updated to the version taken from the same ASF release as that of the example application.

For example, if the TCP client example application is taken from ASF v.3.29.0, the ATWINC1500 should be updated with the firmware available as part of the ATWINC1500 Firmware Update project as shown. The firmware update procedure is described in the ATWINC1500 Getting Started Guide.

Figure 3-4. ATWINC1500 Firmware Update Project



To program the SAM D21 with the TCP client example application,

1. Connect the ATWINC1500 to the EXT1 header of the SAM D21 Xplained Pro board.
2. Connect a micro USB cable to the debug USB header of the SAM D21 Xplained Pro board.
3. Open a serial port terminal application with the COM port configuration 115200,8,None,1,None.
4. Modify the `main.h` of the TCP client application as mentioned in previous sections of this document.
5. Compile and download the application firmware image into the SAM D21 Xplained Pro board.
6. Configure the TCP server as described in the section Testing TCP Client with a standard TCP server or using the TCP Server Example application.

3.4.2. Running the TCP Client Example Application

This section is a quick-start for running the TCP client application. It uses the following hardware as an example:

1. The SAM D21 Xplained Pro
2. The ATWINC1500 connected to EXT1 header of the SAM D21 Xplained Pro

3.5. Testing the TCP Client with a Standard TCP Server

There are several TCP server applications available that are built for Windows®, Android, Python, and so on. Hercules is one such tool that can be setup as a UDP/IP and TCP/IP Client/Server terminal.

1. Connect the PC running Hercules into the same Wi-Fi network as the TCP client
2. Run ipconfig from the PC's commmand prompt and note the IP address of the PC. This shall be set as the `MAIN_WIFI_M2M_SERVER_IP` in the `main.h` of the TCP client example application.
3. Open Hercules -> TCP server tab
4. Set the same port number as set in TCP client example application (`MAIN_WIFI_M2M_SERVER_PORT`)
5. Press the Listen button for the TCP server to start listening on the port.
   Now, execute the TCP client application. The connection will be established and a single packet of data will be sent to the TCP server as shown.

Figure 3-5. TCP Client Application Console Output

3.6. Testing the TCP Client with the ATWINC1500 TCP Server

The TCP server example application is available for ATWINC1500 and can be used as a TCP server. The configuration of the TCP server is explained in Configuration. After the TCP server is setup, the TCP client application can be run to transfer the data to the TCP server.

4. TCP Server Application using ATWINC1500

The TCP server application demonstrates a simple TCP server using a BSD socket to receive and transmit data.

4.1. Application Overview

The TCP server application allows the user to listen for incoming connections, receive data from a TCP client using a socket and transmit data to the TCP client. The reference application implements transfer of a single data packet and this can be extended as per user requirements.

4.2. Configuration

The configuration parameters are present in `main.h` of the TCP server application. The parameters of the Access Point (AP) in the Wi-Fi network, to which the ATWINC1500 joins as a Station (STA) are:

```
#define MAIN_WLAN_SSID   "atwinc1500"    /**< Destination SSID */
#define MAIN_WLAN_AUTH   M2M_WIFI_SEC_WPA_PSK    /**< Security manner */
#define MAIN_WLAN_PSK    "Atmel1234$"   /**< Password for Destination SSID */
```

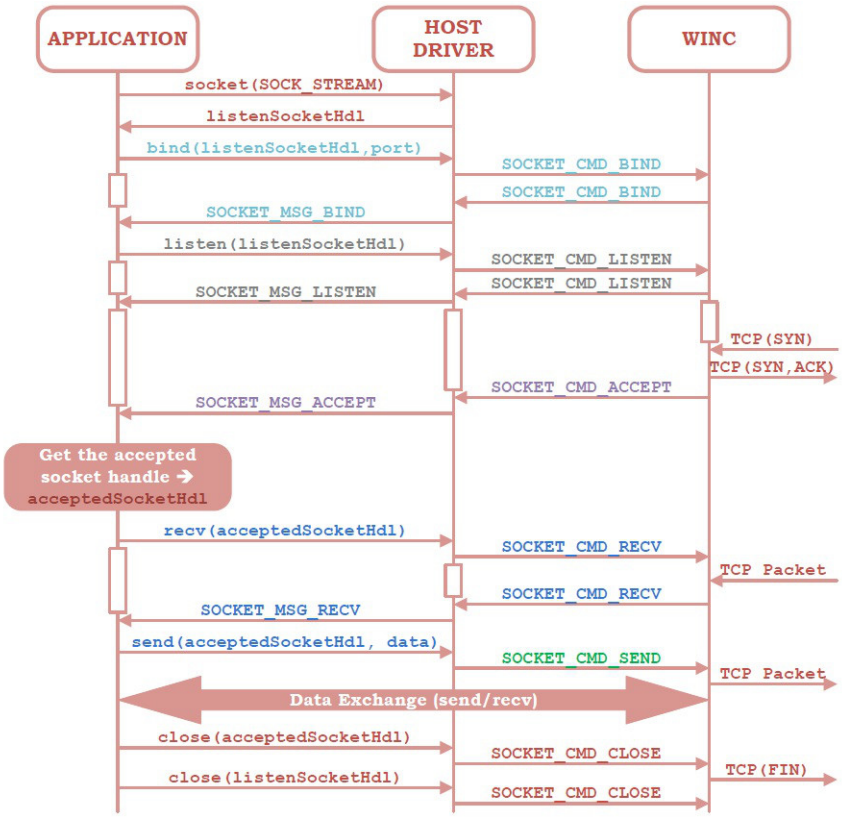
The TCP server socket IP address definition is not used if DHCP is used in the application. The port number should be the same as used in the TCP client application.

```
#define MAIN_WIFI_M2M_SERVER_IP           0xFFFFFFFF /* 255.255.255.255 */
#define MAIN_WIFI_M2M_SERVER_PORT         (6666)
```

4.3. Application Structure

The overall transaction sequence for a TCP server is shown in the following figure.

Figure 4-1. TCP Server Application Flow



The sections describing initialization of the host MCU, Wi-Fi driver and module, interrupt handling, and connection to the AP for the TCP client is the same for the TCP server.

4.3.1. Creating a TCP Server

A TCP server socket is created using the `socket()` API.

The TCP server should be bound to the socket IP address and port using the `bind()` API. Calls to other socket API such as `send()` and `recv()` shall not be done without successful binding. The bind API associates a local network transport address with a socket. It is necessary for a server process to issue an explicit bind request before it can accept connections or start communication with clients.

However, for a client, it is not mandatory to issue a bind call. The ATWINC1500 firmware stack takes care of doing an implicit binding when the client process issues the connect API.

4.3.2. Handling the Socket Event Callback

When the socket event callback is registered using the `registerSocketCallback()` API, socket event notifications are delivered to application through this callback. In the `main.c` file of the TCP server application,

```
static void socket_cb(SOCKET sock, uint8_t u8Msg, void *pvMsg)
{
    switch (u8Msg) {
    /* Socket connected */
    case SOCKET_MSG_CONNECT:
    {
        tstrSocketConnectMsg *pstrConnect = (tstrSocketConnectMsg *)pvMsg;
              ....
       }
       ......
```

```
    }
}
```

There are several useful notifications presented through this socket callback.

Table 4-1. Event Notifications in the Socket Callback Function

msgType	Event
SOCKET_MSG_BIND	This notification is given to the socket callback on completion of socket binding. If the binding has been successful, the TCP server is set to listen for incoming connections.
SOCKET_MSG_LISTEN	This notification is given to the socket call-back with the status of whether the TCP server is ready to accept incoming connections. If the TCP server is ready to listen, it is set to accept new connections.
SOCKET_MSG_ACCEPT	This notification is given to the socket call-back when a new connection is accepted and data is received from the TCP client.

4.3.3. Listening to Incoming TCP Client Connections

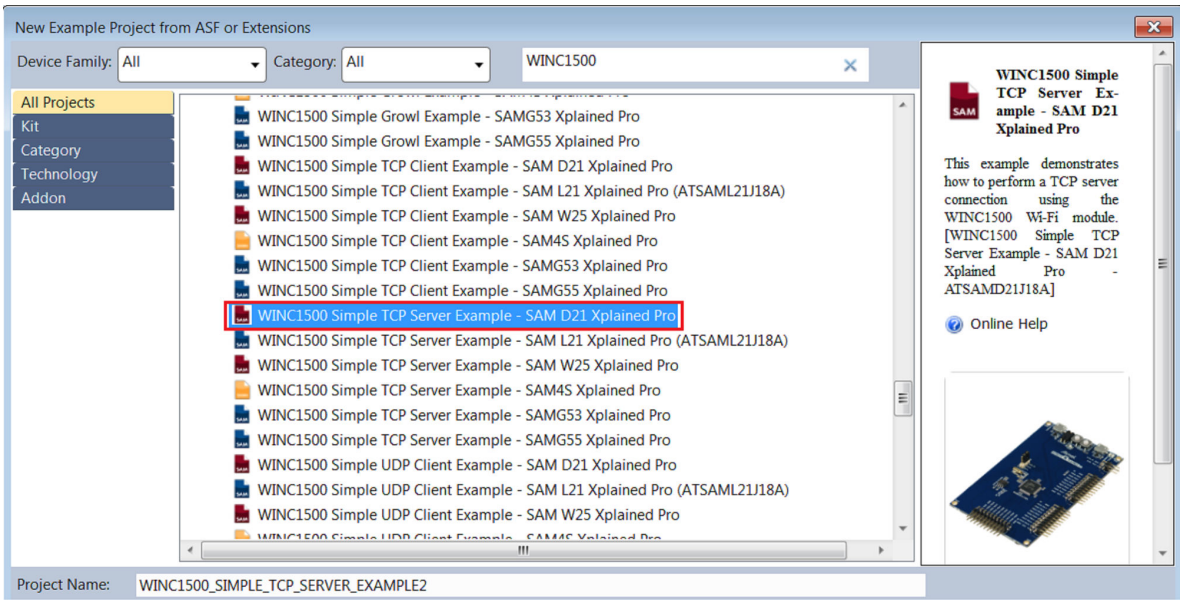
The `listen()` API is used for server operations with TCP stream sockets. After calling the listen API, the socket will automatically accept a connection request from a remote device and the `accept()` API does not have to be called from the application. The listen function causes a `SOCKET_MSG_LISTEN` event notification to be sent to the host after the socket port is ready to indicate listen operation success or failure.

When a remote peer establishes a connection, a `SOCKET_MSG_ACCEPT` event notification is sent to the application. The send and receive functions and associated notifications to the socket callback remain same as for the TCP client.

4.4. Getting Started with the TCP Server Application in ASF

1. Open Atmel Studio 7.0 **File** > **New** > **Example Projects**.
2. Search for WINC1500 in the Search bar on the top right corner. A list of WINC1500 projects would be displayed.
3. Select the WINC1500 Simple TCP Server Example Application for SAM D21 as shown.

Figure 4-2. TCP Server Example Application



The directory structure of the TCP server application is similar to the TCP client application directory structure.

4.4.1. Programming the SAM D21 and ATWINC1500

Before programming the application on the SAM D21 Host MCU, the user must ensure that the firmware on the ATWINC1500 is updated to the version taken from the same ASF release as that of the example application.

For example, if the TCP server example application is taken from ASF v.3.29.0, the ATWINC1500 should be updated with the firmware available as part of the ATWINC1500 Firmware Update Project as shown in Programming the SAM D21 and ATWINC1500. The firmware update procedure is described in the ATWINC1500 Getting Started Guide.

To program the SAM D21 with the TCP server example application,

1. Connect the ATWINC1500 to the EXT1 header of the SAM D21 Xplained Pro board
2. Connect a micro USB cable to the debug USB header of the SAM D21 Xplained Pro board
3. Open a serial port terminal application with the COM port configuration 115200,8,None,1,None
4. Modify the `main.h` of the TCP server application as mentioned in previous sections of this document.
5. Compile and download the application firmware image into the SAM D21 Xplained Pro board.
6. Configure the TCP client as described in Configuration.

4.4.2. Running the TCP Server Example Application

This section is a quick-start for running the TCP server application. It uses the following hardware as an example:

1. The SAM D21 Xplained Pro
2. The ATWINC1500 connected to EXT1 header of the SAM D21 Xplained Pro

4.5. Testing the TCP Server with a Standard TCP Client

This section uses Hercules as the TCP client.

1. Connect the PC running Hercules into the same Wi-Fi network as the TCP server.
2. Open **Hercules** -> **TCP client** tab
3. Note the TCP server IP address from the console terminal of the TCP server. Enter this in Hercules in the module IP box.
4. Set the same port number as set in TCP client example application (`MAIN_WIFI_M2M_SERVER_PORT`)
5. Press the connect button for the TCP client to establish connection with the TCP server.

The connection will be established and a single packet of data will be sent from and to the TCP server as shown in the following figure.

Figure 4-3. TCP Server Application Console Output

```
-- TCP server example --
-- SAMD21_XPLAINED_PRO --
-- Compiled: xxx xx xxxx xx:xx:xx –
wifi_cb: M2M_WIFI_RESP_CON_STATE_CHANGED: CONNECTED
wifi_cb: M2M_WIFI_REQ_DHCP_CONF: IP is xxx.xxx.xxx.xxx
socket_cb: bind success!
socket_cb: listen success!
socket_cb: accept success!
socket_cb: recv success!
socket_cb: send success!
TCP Server Test Complete!
close socket
```

4.6. Testing the TCP Server with the ATWINC1500 TCP Client

This scenario can be tested as described in Getting Started with the TCP Client Application in ASF.

5. Revision History

Doc. Rev.	Date	Comments
42739A	08/2016	Initial document release.

Atmel | Enabling Unlimited Possibilities®

Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA **T:** (+1)(408) 441.0311 **F:** (+1)(408) 436.4200 | **www.atmel.com**