User's Guide

Version 1.0

May 10, 2013

CC3000 - Host Driver - Multithread Support

Contents

Introduction

This document lists down the instructions for developing single threaded and multithreaded applications using the CC3000 host-driver. It must be read in conjunction with “CC3000 Host-Driver Multithread-Support Approach Note” and “TI SYS/BIOS v6.34 User Guide”.

Instructions for installing TI-RTOS can be found in TI_RTOS_GettingStartedGuide.

Software Structure

Applications, both single-threaded and multithreaded, can be developed with the same API set. The host-driver package consists of multiple components, which must be linked together for achieving the desired functionality. The details are explained in subsequent sections.

- CC300 Host Driver APIs: The user APIs are divided into four silos to reflect the four different entities that correspond within the device. CC3000_Host_Programming_Guide has detailed explanation on the topic
- Single-threaded library: It has the wrappers to the core-library functions
- Multithreaded library: It synchronizes multiple user threads from accessing core-library functions. The architectural details are explained in “CC3000 Host Driver Multithread Support Approach Note”
- Core Library + HCI Abstraction: CC3000_Host_Programming_Guide has detailed explanation on the topic
- SPI Transport Layer: CC3000_Host_Programming_Guide has detailed explanation on the topic
- OSAL: Operating System Abstraction Layer

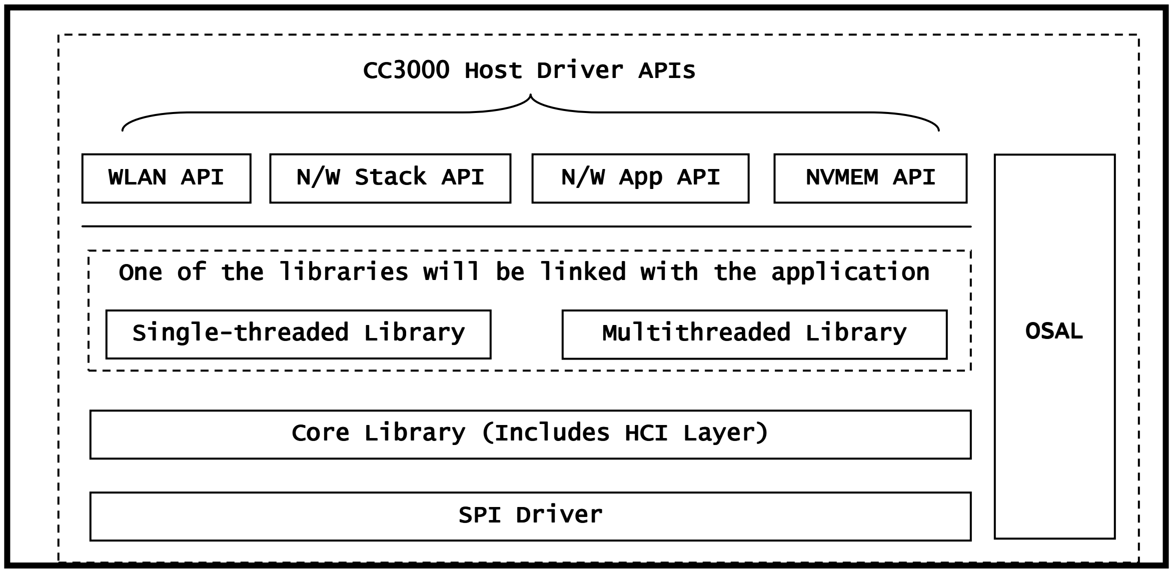


Diagram 1: Software Structure

Single threaded applications

Below components must be added to the project/workspace for executing legacy applications or developing new single-threaded applications.

- Binaries:
  - mt_cc3000_host_driver.lib ← `CC3000 Host Driver`
  - mt_cc3000_spi.lib ← `SPI Driver`

'__ENABLE_MULTITHREADED_SUPPORT__' must be undefined when building above binaries.

Multithreaded applications

Below components must be added to the project for developing new multithreaded applications:

- Binaries:
  - mt_impl_multi_thread.lib ← `Multithreaded implementation`
  - mt_cc3000_host_driver.lib ← `Host Driver`
  - mt_cc3000_spi.lib ← `SPI Driver`
  - mt_osal.lib ← `OSAL`
  - mt_wifi_app_multi_thread.out ← `Multithreaded Application`

"mt_impl_multi_thread.lib" defines the user APIs. It synchronizes multiple user threads from accessing core-library functions. The resulting behavior is function-specific and determined by the definition of the function and the context to which it is applied.

'__ENABLE_MULTITHREADED_SUPPORT__' must be defined when building above binaries.

```
int socket(long domain, long type, long protocol)
{
   int ret;
   OS_mutex_lock(g_main_mutex, &mtx_key);
   ret = c_socket(domain, type, protocol);
   if (-1 != ret) find_add_next_free_socket(ret);
   OS_mutex_unlock(g_main_mutex, mtx_key);
   return(ret);
}
```

Code Snippet 3 – "socket"

```
int recv(long sd, void *buf, long len, long flags)
{
   int ret = 0, index = 0;
   OS_semaphore_post(g_select_sleep_semaphore); //wakeup select thread if needed
   for (index = 0; index < MAX_NUM_OF_SOCKETS; index++){
      if (g_sockets[index].sd == sd){
         OS_semaphore_pend(g_sockets[index].sd_semaphore, e_WAIT_FOREVER); }
   }
   OS_semaphore_pend(g_select_sleep_semaphore, e_WAIT_FOREVER);
   if (g_wlan_stopped) return -1;
   OS_mutex_lock(g_main_mutex, &mtx_key);
   ret = c_recv(sd, buf, len, flags);
   OS_mutex_unlock(g_main_mutex, mtx_key);
   return(ret);
}
```

Code Snippet 4 – "recv"

Sample application

A sample application is provided with the package to showcase the multithreaded host-driver capabilities. It's a client-server application that transmits and receives data in parallel. It spawns two tasks when launched, a socket-server (call it 'rx_thread') and a socket-client (call it 'tx_thread'), one waiting for data from an 'iperf-client' and the other sending data to an 'iperf-server' respectively.

The 'tx_thread' runs every time two hundred packets are received by an 'rx_thread' and transmits fifty packets to the 'iperf_server'. It again waits for 'rx_thread' to receive predefined number of packets.

Details of applications' definition are listed in "CC3000 Host Driver Multithread Support Approach Note".

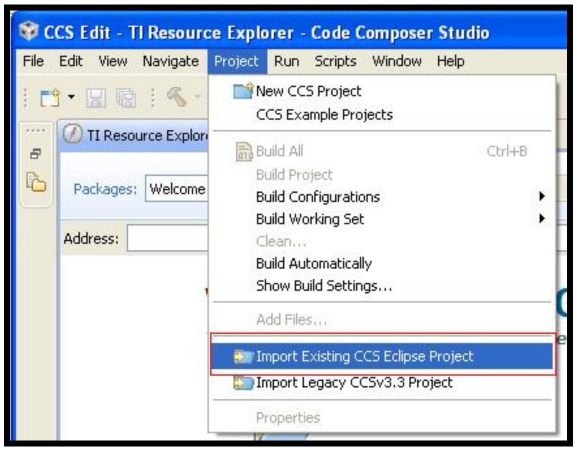
Build and Execute

CCS v5.3

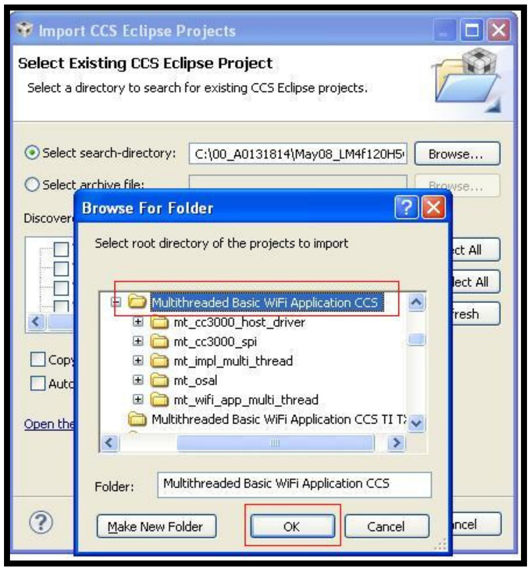
The ‘Multithread Wi-Fi Application’ is supported on CCS workbench only.

Environment Setup

- Open CCS v5
- Set the workspace
- Importing CCS project from workspace directory
    - Select “Project->Import Existing CCS/CCE Eclipse Project”.

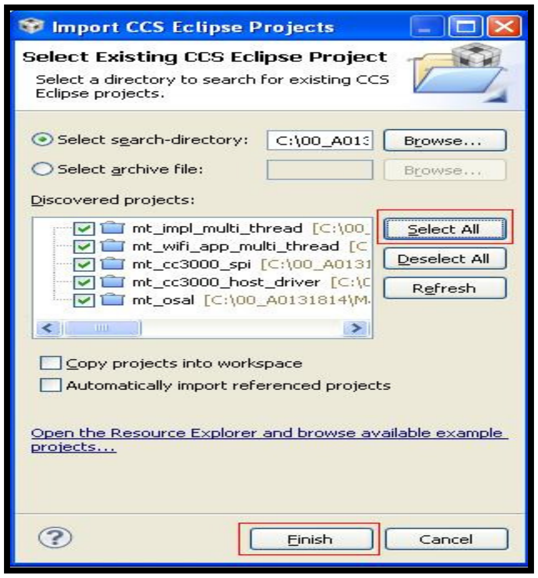


    - Select “Browse” and open “Multithreaded_WiFi_Application_CCS” folder.


Import CCS Eclipse Projects
Select Existing CCS Eclipse Project
Select a directory to search for existing CCS Eclipse projects.
Select search-directory:
C:\00_A0131814\May08_LM4f120H5
Browse...
Select archive file:
Browse...
Discover
Copy
Auto
Open the
Browse For Folder
Select root directory of the projects to import
Multithreaded Basic WiFi Application CCS
mt_cc3000_host_driver
mt_cc3000_spi
mt_impl_multi_thread
mt_osal
mt_wifi_app_multi_thread
Multithreaded Basic WiFi Application CCS TI T
Folder:
Multithreaded Basic WiFi Application CCS
Make New Folder
OK
Cancel

- Select “Select All” and “Finish”

  Note: ‘Copy projects into workspace’ shouldn’t be ticked



Import CCS Eclipse Projects

Select Existing CCS Eclipse Project

Select a directory to search for existing CCS Eclipse projects.

(●) Select search-directory: C:\00_A013 [Browse...]

( ) Select archive file: [Browse...]

Discovered projects:

- [✓] mt_impl_multi_thread [C:\00_
- [✓] mt_wifi_app_multi_thread [C
- [✓] mt_cc3000_spi [C:\00_A0131
- [✓] mt_cc3000_host_driver [C:\C
- [✓] mt_osal [C:\00_A0131814\M.

[ ] Copy projects into workspace

[ ] Automatically import referenced projects

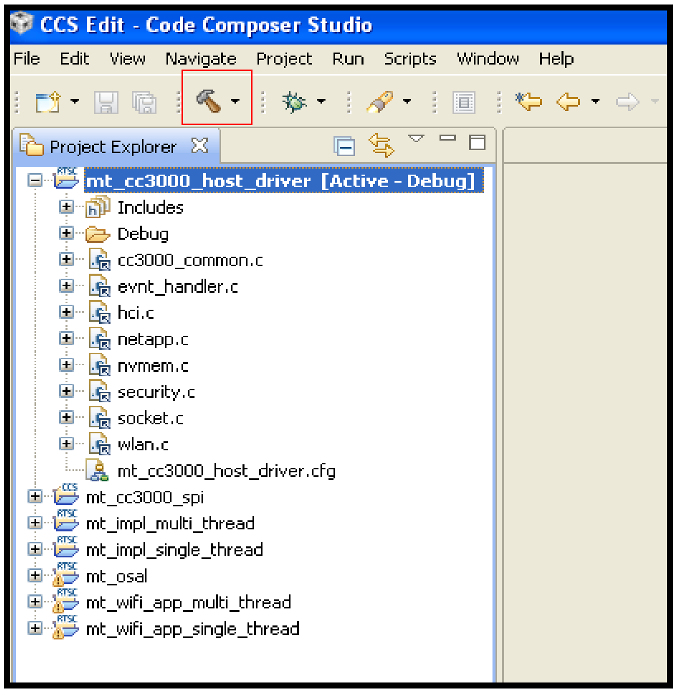
Open the Resource Explorer and browse available example projects...

- At this step a workspace will be created

Building the multithreaded application

In the workspace window, select the project that needs to be build and press hammer icon for building it.

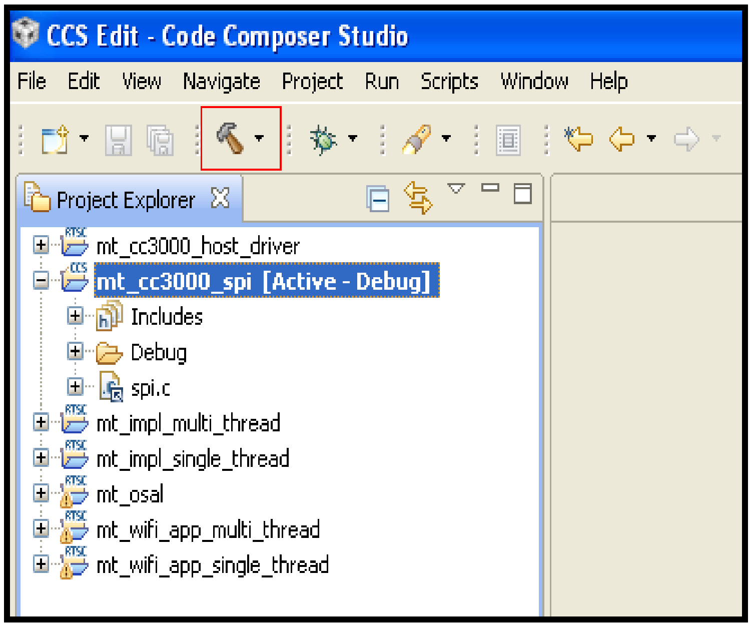
- Building “mt_cc3000_host_driver”



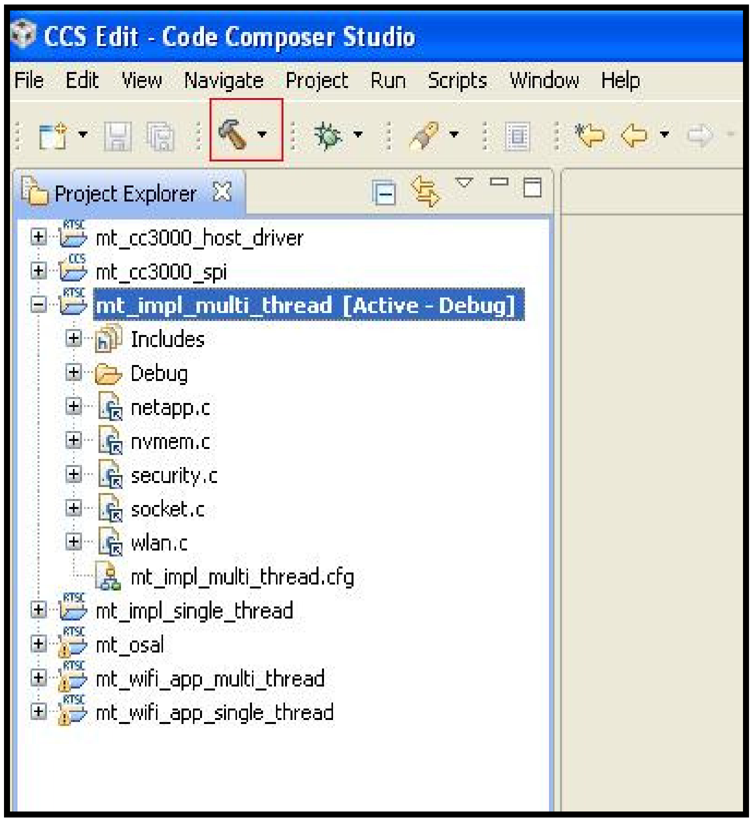
CCS Edit - Code Composer Studio
File Edit View Navigate Project Run Scripts Window Help
Project Explorer

- mt_cc3000_host_driver [Active - Debug]
  - Includes
  - Debug
  - cc3000_common.c
  - evnt_handler.c
  - hci.c
  - netapp.c
  - nvmem.c
  - security.c
  - socket.c
  - wlan.c
  - mt_cc3000_host_driver.cfg
- mt_cc3000_spi
- mt_impl_multi_thread
- mt_impl_single_thread
- mt_osal
- mt_wifi_app_multi_thread
- mt_wifi_app_single_thread

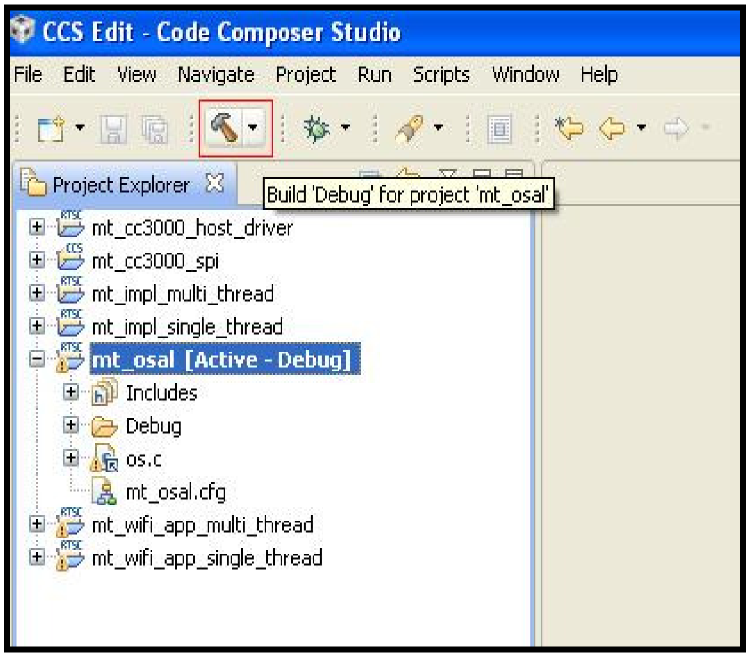
- Building “mt_cc3000_spi”


CCS Edit - Code Composer Studio
File Edit View Navigate Project Run Scripts Window Help
Project Explorer
mt_cc3000_host_driver
mt_cc3000_spi [Active - Debug]
Includes
Debug
spi.c
mt_impl_multi_thread
mt_impl_single_thread
mt_osal
mt_wifi_app_multi_thread
mt_wifi_app_single_thread

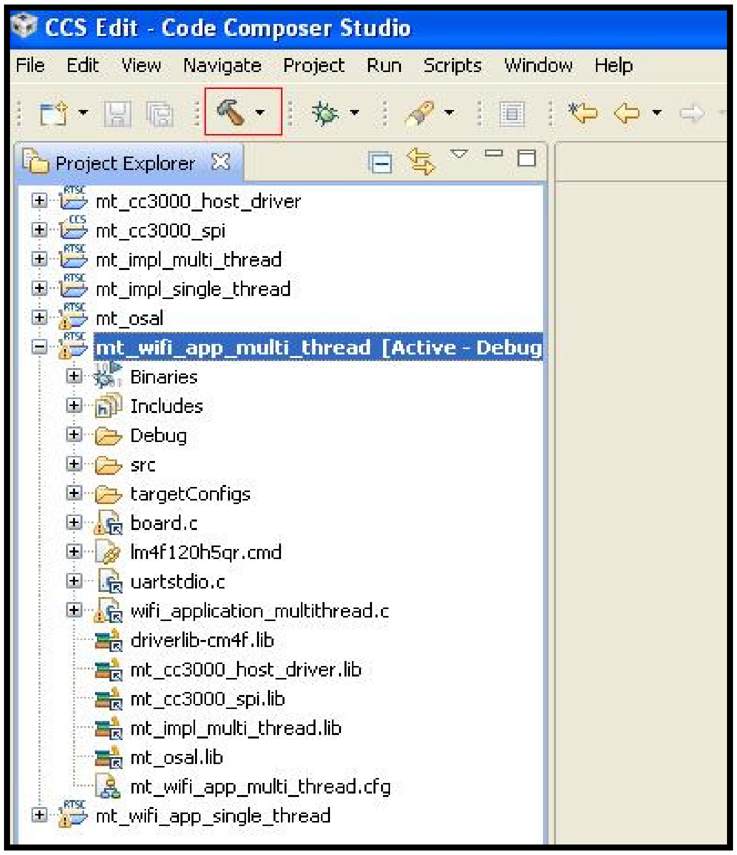
- Building “mt_impl_multi_thread”



- Building “mt_osal”



- Building “mt_wifi_app_multi_thread”



Running the multithreaded application

The example application demonstrates basic WLAN connectivity together with the capability of sending and receiving data over the WLAN transport with the help of the standard BSD socket interface.

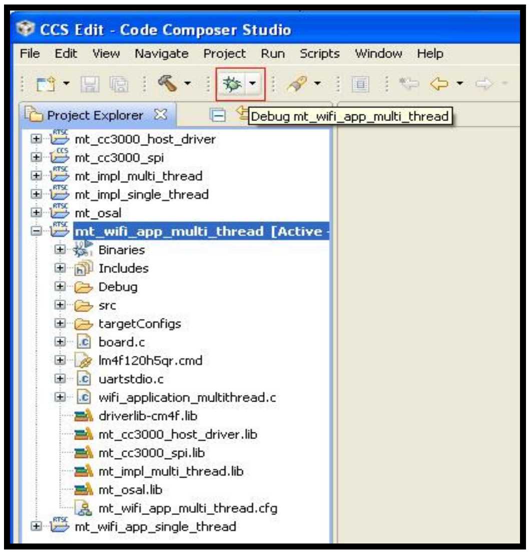
Multithreaded application

- Configure “iperf” as a server listening on port 9002 by executing below command

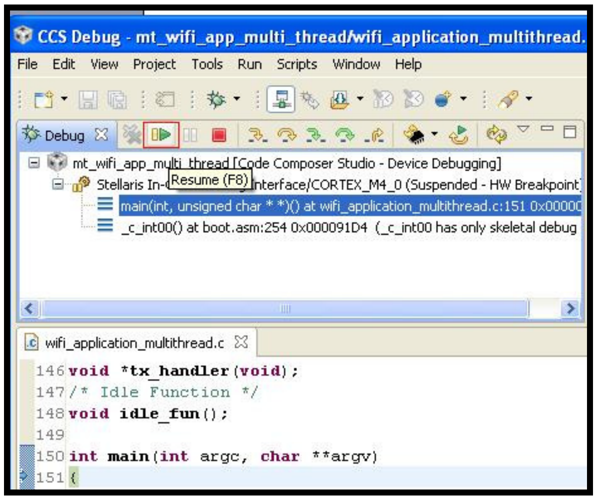
```
iperf.exe -s -p 9002 -i 1 -u
```

Note: Ensure antivirus is disabled on the host-PC for ‘iperf’ to work smoothly

- Set up the board and download the image


CCS Edit - Code Composer Studio
File Edit View Navigate Project Run Scripts Window Help
Project Explorer
Debug mt_wifi_app_multi_thread
mt_cc3000_host_driver
mt_cc3000_spi
mt_impl_multi_thread
mt_impl_single_thread
mt_osal
mt_wifi_app_multi_thread [Active
Binaries
Includes
Debug
src
targetConfigs
board.c
lm4f120h5qr.cmd
uartstdio.c
wifi_application_multithread.c
driverlib-cm4f.lib
mt_cc3000_host_driver.lib
mt_cc3000_spi.lib
mt_impl_multi_thread.lib
mt_osal.lib
mt_wifi_app_multi_thread.cfg
mt_wifi_app_single_thread

- Run the sample application


CCS Debug - mt_wifi_app_multi_thread/wifi_application_multithread.
File Edit View Project Tools Run Scripts Window Help
Debug
Resume (F8)
wifi_application_multithread.c

mt_wifi_app_multi_thread [Code Composer Studio - Device Debugging]
Stellaris In-Circuit Debug Interface/CORTEX_M4_0 (Suspended - HW Breakpoint)
main(int, unsigned char * *)() at wifi_application_multithread.c:151 0x0000C
_c_int00() at boot.asm:254 0x000091D4 (_c_int00 has only skeletal debug

```
void *tx_handler(void);
/* Idle Function */
void idle_fun();

int main(int argc, char **argv)
{
```

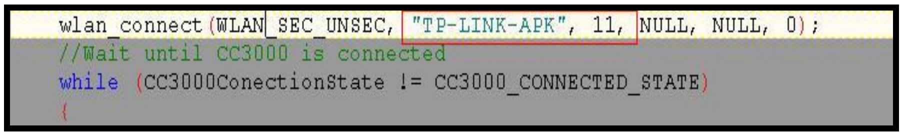
- WLAN is turned ON and device gets connected to a network.

  - Connection using Smart-Configuration: It's the default connection mode of the multithreaded application. The device can be connected to the AP using Texas Instruments' 'SmartConfig^{TM}' application. Detailed instructions can be found in CC3000_Smart_Config_Wiki. A blinking red LED indicates an ongoing 'Smart Configuration' process. The application does not support AES encryption enabled 'SmartConfig^{TM}' mode

  - Connection using 'wlan_connect' API – Connection can be established with the AP using 'wlan_connect' API of 'mt_impl_multi_thread.lib'.

    - Do not define 'ENABLE_SMART_CONFIG' when building 'mt_wifi_app_multi_thread.out'

    - Second and third argument to 'wlan_connect' API is 'SSID Name' and 'Length of SSID Name'. Change the values to establish connection with corresponding AP



```
wlan_connect(WLAN_SEC_UNSEC, "TP-LINK-APK", 11, NULL, NULL, 0);
//Wait until CC3000 is connected
while (CC3000ConectionState != CC3000_CONNECTED_STATE)
{
```

- "rx-thread-1" and "rx-thread-2" are spawned – These threads implement socket-servers and waits for clients to connect and send data on port 9000 and 9001 respectively.

- "tx_thread" is spawned – It connects to the "iperf-server" and sends fifty data packets on port 9002 whenever rx_threads receive two hundred packets from their respective clients. "iperf" server logs the received data on its terminal.

  Note: Destination address and port is currently hardcoded in the application. Destination address has to be modified and the application must be rebuilt for connecting with a server and sending data to it.

Filename: wifi_application_multithread.c

```
#define HOST_NAME <addess>
void *tx_handler(void)
{
    …
    …
    srvr_addr.sa_data[2] = <octet1>;  /* First Octet of destination IP */
    srvr_addr.sa_data[3] = <octet2>;  /* Second Octet of destination IP */
    srvr_addr.sa_data[4] = <octet3>;  /* Third Octet of destination IP */
    srvr_addr.sa_data[5] = <octet4>;  /* Fourth octet of destination IP */
}
```

For ex: If data is intended to be sent to server with IP Address 192.168.1.100, then:

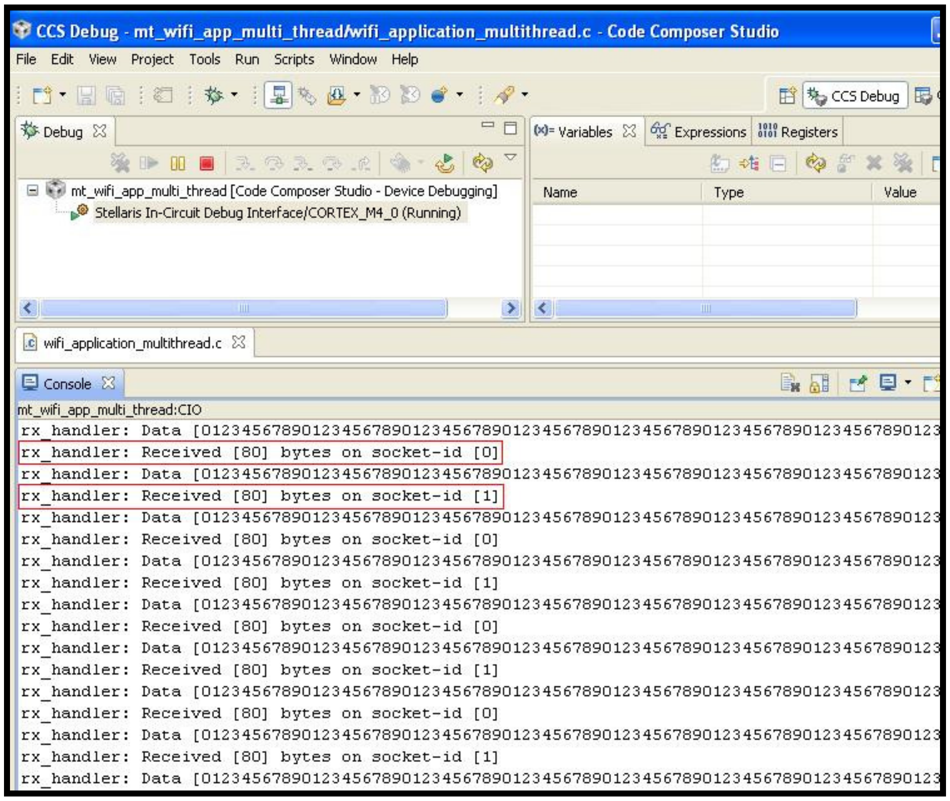
- srvr_addr.sa_data[2] = 192
- srvr_addr.sa_data[3] = 168
- srvr_addr.sa_data[4] = 1
- srvr_addr.sa_data[5] = 100

- Configure “iperf” as a client sending data on port 9000 and 9001 by executing below commands on two separate terminals.

```
iperf.exe -c [IP Address] -p [Port] -i 1 -u
```

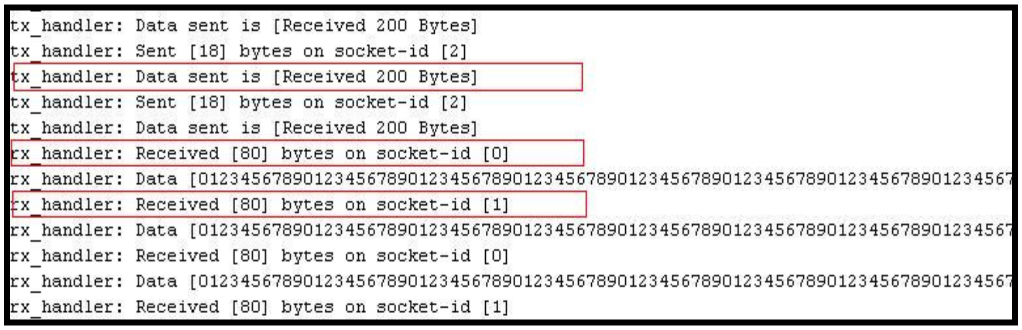
- Data received from “iperf” clients can be seen printed on the workbench’s console.

  Note: CC3000 device and “iperf’s” workstation should be connected to the same network


CCS Debug - mt_wifi_app_multi_thread/wifi_application_multithread.c - Code Composer Studio
File Edit View Project Tools Run Scripts Window Help
CCS Debug
Debug
Variables
Expressions
Registers
mt_wifi_app_multi_thread [Code Composer Studio - Device Debugging]
Stellaris In-Circuit Debug Interface/CORTEX_M4_0 (Running)
Name
Type
Value
wifi_application_multithread.c
Console

```
mt_wifi_app_multi_thread:CIO
rx_handler: Data [01234567890123456789012345678901234567890123456789012345678901234567890123
rx_handler: Received [80] bytes on socket-id [0]
rx_handler: Data [01234567890123456789012345678901234567890123456789012345678901234567890123
rx_handler: Received [80] bytes on socket-id [1]
rx_handler: Data [01234567890123456789012345678901234567890123456789012345678901234567890123
rx_handler: Received [80] bytes on socket-id [0]
rx_handler: Data [01234567890123456789012345678901234567890123456789012345678901234567890123
rx_handler: Received [80] bytes on socket-id [1]
rx_handler: Data [01234567890123456789012345678901234567890123456789012345678901234567890123
rx_handler: Received [80] bytes on socket-id [0]
rx_handler: Data [01234567890123456789012345678901234567890123456789012345678901234567890123
rx_handler: Received [80] bytes on socket-id [1]
rx_handler: Data [01234567890123456789012345678901234567890123456789012345678901234567890123
rx_handler: Received [80] bytes on socket-id [0]
rx_handler: Data [01234567890123456789012345678901234567890123456789012345678901234567890123
rx_handler: Received [80] bytes on socket-id [1]
rx_handler: Data [01234567890123456789012345678901234567890123456789012345678901234567890123
```

- Below snapshot shows all three thread in action


```
tx_handler: Data sent is [Received 200 Bytes]
tx_handler: Sent [18] bytes on socket-id [2]
tx_handler: Data sent is [Received 200 Bytes]
tx_handler: Sent [18] bytes on socket-id [2]
tx_handler: Data sent is [Received 200 Bytes]
rx_handler: Received [80] bytes on socket-id [0]
rx_handler: Data [01234567890123456789012345678901234567890123456789012345678901234567
rx_handler: Received [80] bytes on socket-id [1]
rx_handler: Data [01234567890123456789012345678901234567890123456789012345678901234567
rx_handler: Received [80] bytes on socket-id [0]
rx_handler: Data [01234567890123456789012345678901234567890123456789012345678901234567
rx_handler: Received [80] bytes on socket-id [1]
```

SYS/BIOS Configuration

The application is developed on TI SYS/BIOS v6.34 RTOS – Please refer Chapter 2 of TI SYS/BIOS v6.34 User Guide for detailed instruction on creating a new SYS/BIOS project or adding SYS/BIOS support to an existing project.

SYS/BIOS configuration for sample application:

- SYS/BIOS modules used by the application

```
var ti_sysbios_BIOS = xdc.useModule('ti.sysbios.BIOS');
var Task = xdc.useModule('ti.sysbios.knl.Task');
var Idle = xdc.useModule('ti.sysbios.knl.Idle');
var Hwi = xdc.useModule('ti.sysbios.hal.Hwi');
var Error = xdc.useModule('xdc.runtime.Error');
var TIRTOS = xdc.useModule('ti.tirtos.TIRTOS');
var GPIO = xdc.useModule('ti.drivers.GPIO');
var Defaults = xdc.useModule('xdc.runtime.Defaults');
var Text = xdc.useModule('xdc.runtime.Text');
var System = xdc.useModule('xdc.runtime.System');
var SysMin = xdc.useModule('xdc.runtime.SysMin');
var Error = xdc.useModule('xdc.runtime.Error');
```

- Only the modules and APIs used by the application are included

```
ti_sysbios_BIOS.libType = ti_sysbios_BIOS.LibType_Custom;
```

- Minimizes the memory requirements of a SYS/BIOS application

```
Program.argSize = 0x4;
Defaults.common$.namedModule = false;
Text.isLoaded = false;
System.maxAtexitHandlers = 0;
ti_sysbios_BIOS.swiEnabled = false;
ti_sysbios_BIOS.clockEnabled = false;
SysMin.flushAtExit = false;
System.SupportProxy = SysMin;
Error.raiseHook = null;
ti_sysbios_BIOS.rtsGateType = ti_sysbios_BIOS.NoLocking;
```

- Creating ‘launcher” service

```
var task0Params = new Task.Params();
task0Params.instance.name = "launcher";
task0Params.vitalTaskFlag = false;
task0Params.priority = 3;
task0Params.stackSize = 0x400;
Program.global.server = Task.create("&launch_service", task0Params);
```