Application Note

Implementing a FTP Server on TI 66AK2H Device With RTOS

Eric Ding

ABSTRACT

This application report shows how to implement a FTP server example on TI KeyStone II 66AK2H device. The example is built with TI Processor SDK RTOS and NDK packages. While the note explains the software porting and development procedures, it focuses on methods how to tune up the network throughput. Those tuning methods are applicable to other TI processors and common TCP/IP networks in general.

Table of Contents

List of Figures

Trademarks

Code Composer Studio™ is a trademark of Texas Instruments.
Arm® and Cortex® are registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere.
All trademarks are the property of their respective owners.

1 Introduction

1.1 TI Processor SDK RTOS

The Processor Software Development Kit (Processor SDK) provides the core foundation and building blocks that facilitate application software development on TI's embedded processors. The package consists of device and platform dependent modules, core dependent software with source code and prebuilt libraries, TI Real-Time Operating System (RTOS) kernel, utilities and application examples. The TI Processor SDK RTOS is available as a free download with all components in one installer.

1.2 TI NDK

The Network Development Kit (NDK) https://software-dl.ti.com/processor-sdk-rtos/esd/docs/latest/rtos/index_Foundational_Components.html#ndk is a platform for development and demonstration of network enabled RTOS applications on TI processors and includes demonstration software showcasing capabilities across a range of network enabled applications.

The NDK provides an IPv6 and IPv4 compliant TCP/IP stack working with the TI-RTOS Kernel. It primarily focuses on providing the core Layer 3 and Layer 4 stack services along with additional higher-level network applications. The NDK itself does not include any platform or device specific software. The NDK interfaces through well-defined transport interface, Network Interface Management Unit (NIMU), to the PDK and platform software elements needed for operation.

1.3 66AK2H Device

The 66AK2Hxx (a.k.a. K2H) platform combines four Arm® Cortex®-A15 processors with up to eight TMS320C66x high-performance DSPs using the KeyStone II architecture. K2H device is optimal for embedded infrastructure applications like cloud computing, media processing, high-performance computing and so forth. The K2H has a five-port Gigabit Ethernet switch, with one port to host, and four Serial Gigabit Media Independent Interface (SGMII) ports connected to the networks.

1.4 FTP Offering in TI Processor SDK RTOS

The File Transfer Protocol (FTP) is a common network protocol used for file transfer between a client and a server structure. The FTP uses TCP protocol for reliable data transfer. As seen from the TI Processor SDK RTOS user guide: https://software-dl.ti.com/processor-sdk-rtos/esd/docs/latest/rtos/index_Foundational_Components.html#id140, an FTP server example is available to some processors, but not including K2H.

As a multi-core high performance processor, it is desired to have such an example on the K2H device for customer network application reference. Porting the existing one to K2H is expected to be straightforward. The note explains how to create a Code Composer Studio™ (CCS) project, port the code and test. Moreover, network performance is very important in real life. Such discussions may spread over the TI (E2E forums) or wider Internet communities. The application report examines how to optimize the network throughput from various approaches and summarizes what we learnt from this practice.

2 Hardware and Software

The K2H EVM () is used for the FTP server development and test. The EVM provides connectivity for the four SGMII Gigabit Ethernet ports:

- An Ethernet PHY is connected to SGMII ports 0 and 1, respectively, to provide a copper interface and routed to a Gigabit RJ-45 connector
- The SGMII ports 2 and 3 are routed to ports 0 and 1 of the AMC edge connector backplane interface

In the test, the SGMII port 0 is used by directly connecting to a Gigabit Ethernet port of a Windows 10 PC. The software configures the K2H with a fixed IP address 192.168.1.4 by default, the Windows PC has to be on the same subnet 192.168.1.xxx.

The software installer is downloaded from https://software-dl.ti.com/processor-sdk-rtos/esd/K2HK/latest/index_FDS.html. As time of the development, the latest 06_03_00_106 release is used. The installer contains all the packages necessary for the work, except that TI Code Composer Studio (CCS) needs to be downloaded separately. The packages used in the work are listed as follows:

- SYSBIOS: 6.76.03.01
- XDCTools: 3.55.02.22
- PDK K2HK: 4.0.16
- NDK: 3.61.01.01 + NS: 2.60.01.06
- UIA: 2.30.01.02
- Arm Compiler: arm-none-eabi-gcc-7.2.1 (GNU Tools for Arm Embedded Processors 7-2017-q4-major)
- CCS 9.3.0.00012

3 Develop the FTP Server on K2H

3.1 Reference FTP Server Example

A reference FTP server example called NIMU_FtpExample_<board>_<core>Exampleproject is available inside pdk_k2hk_4_0_xx. As explained in the *PDK Example and Test Project Creation* section in https://software-dl.ti.com/processor-sdk-rtos/esd/docs/latest/rtos/index_overview.html#rebuild-pdk, the CCS project can be created using batch script:

```
pdkProjectCreate [soc] [board] [endian] [module] [project type] [processor] [pdkDir]
```

where module is nimu.

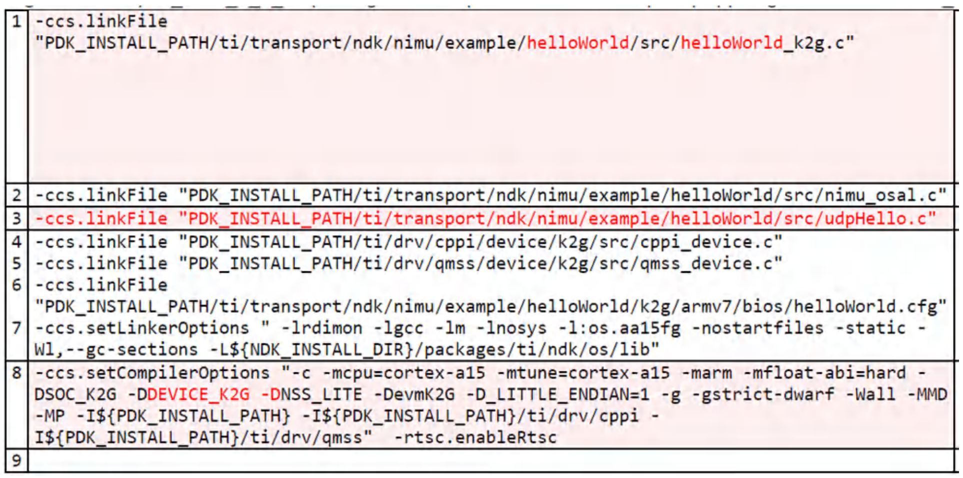
The script searches through the pdk_k2hk_4_0_xx\packages\ti\transport\ndk\nimu (refer to as NIMU_INSTALL_DIR hereafter for brevity) folder for the existence of NIMU_FtpExample_<board>_<core>Exampleproject.txt to create the CCS project. The FTP example is available for several devices, like AM335x, AM437x, K2G and so forth, but not for K2H. The K2G is another KeyStone II processor with ARM A15 and C66x cores, the K2G project can be used as a reference for K2H because of the similarity between the two processors. As an example, file NIMU_INSTALL_DIR \example\ftpApp\k2g\armv7\bios\NIMU_FtpExample_evmK2G_armExampleproject.txt shows all the files, compiler and linker setups for the CCS project on Arm A15.

3.2 Create K2H FTP Server Example

The K2G device has a FTP server example and a basic hello world example, while the K2H device only has the latter. One may compare the two K2G projects for differences, then apply those to the K2H using the hello world example as the template. The note illustrates the steps on how to create the Arm A15 project. The same applies to the C66x project.

As shown below between two .txt files for K2G:

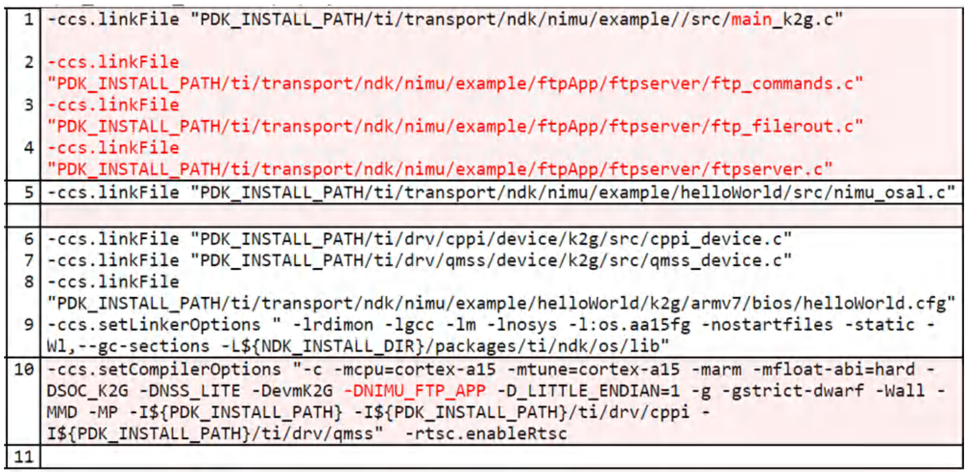
- NIMU_INSTALL_DIR\example\helloWorld\k2g\armv7\bios\NIMU_BasicExample_evmK2G_armExampleproject.txt



```
-ccs.linkFile
"PDK_INSTALL_PATH/ti/transport/ndk/nimu/example/helloWorld/src/helloWorld_k2g.c"
-ccs.linkFile "PDK_INSTALL_PATH/ti/transport/ndk/nimu/example/helloWorld/src/nimu_osal.c"
-ccs.linkFile "PDK_INSTALL_PATH/ti/transport/ndk/nimu/example/helloWorld/src/udpHello.c"
-ccs.linkFile "PDK_INSTALL_PATH/ti/drv/cppi/device/k2g/src/cppi_device.c"
-ccs.linkFile "PDK_INSTALL_PATH/ti/drv/qmss/device/k2g/src/qmss_device.c"
-ccs.linkFile
"PDK_INSTALL_PATH/ti/transport/ndk/nimu/example/helloWorld/k2g/armv7/bios/helloWorld.cfg"
-ccs.setLinkerOptions " -lrdimon -lgcc -lm -lnosys -l:os.aa15fg -nostartfiles -static -
Wl,--gc-sections -L${NDK_INSTALL_DIR}/packages/ti/ndk/os/lib"
-ccs.setCompilerOptions "-c -mcpu=cortex-a15 -mtune=cortex-a15 -marm -mfloat-abi=hard -
DSOC_K2G -DDEVICE_K2G -DNSS_LITE -DevmK2G -D_LITTLE_ENDIAN=1 -g -gstrict-dwarf -Wall -MMD
-MP -I${PDK_INSTALL_PATH} -I${PDK_INSTALL_PATH}/ti/drv/cppi -
I${PDK_INSTALL_PATH}/ti/drv/qmss"  -rtsc.enableRtsc
```

Figure 3-1. K2G Hello World Project File

- NIMU_INSTALL_DIR\example\ftpApp\k2g\armv7\bios\NIMU_FtpExample_evmK2G_armExampleproject.txt



```
-ccs.linkFile "PDK_INSTALL_PATH/ti/transport/ndk/nimu/example//src/main_k2g.c"

-ccs.linkFile
"PDK_INSTALL_PATH/ti/transport/ndk/nimu/example/ftpApp/ftpserver/ftp_commands.c"
-ccs.linkFile
"PDK_INSTALL_PATH/ti/transport/ndk/nimu/example/ftpApp/ftpserver/ftp_filerout.c"
-ccs.linkFile
"PDK_INSTALL_PATH/ti/transport/ndk/nimu/example/ftpApp/ftpserver/ftpserver.c"
-ccs.linkFile "PDK_INSTALL_PATH/ti/transport/ndk/nimu/example/helloWorld/src/nimu_osal.c"

-ccs.linkFile "PDK_INSTALL_PATH/ti/drv/cppi/device/k2g/src/cppi_device.c"
-ccs.linkFile "PDK_INSTALL_PATH/ti/drv/qmss/device/k2g/src/qmss_device.c"
-ccs.linkFile
"PDK_INSTALL_PATH/ti/transport/ndk/nimu/example/helloWorld/k2g/armv7/bios/helloWorld.cfg"
-ccs.setLinkerOptions " -lrdimon -lgcc -lm -lnosys -l:os.aa15fg -nostartfiles -static -
Wl,--gc-sections -L${NDK_INSTALL_DIR}/packages/ti/ndk/os/lib"
-ccs.setCompilerOptions "-c -mcpu=cortex-a15 -mtune=cortex-a15 -marm -mfloat-abi=hard -
DSOC_K2G -DNSS_LITE -DevmK2G -DNIMU_FTP_APP -D_LITTLE_ENDIAN=1 -g -gstrict-dwarf -Wall -
MMD -MP -I${PDK_INSTALL_PATH} -I${PDK_INSTALL_PATH}/ti/drv/cppi -
I${PDK_INSTALL_PATH}/ti/drv/qmss"  -rtsc.enableRtsc
```

Figure 3-2. K2G FTP Server Project File

It is evident that:

- The main source files are different
- The FTP server example has three FTP related source files under ftpserver folder, replacing the udpHello.c for the hello world example.
- The pre-defined symbols are different: -DNIMU_FTP_APP is an addition to the FTP server example .

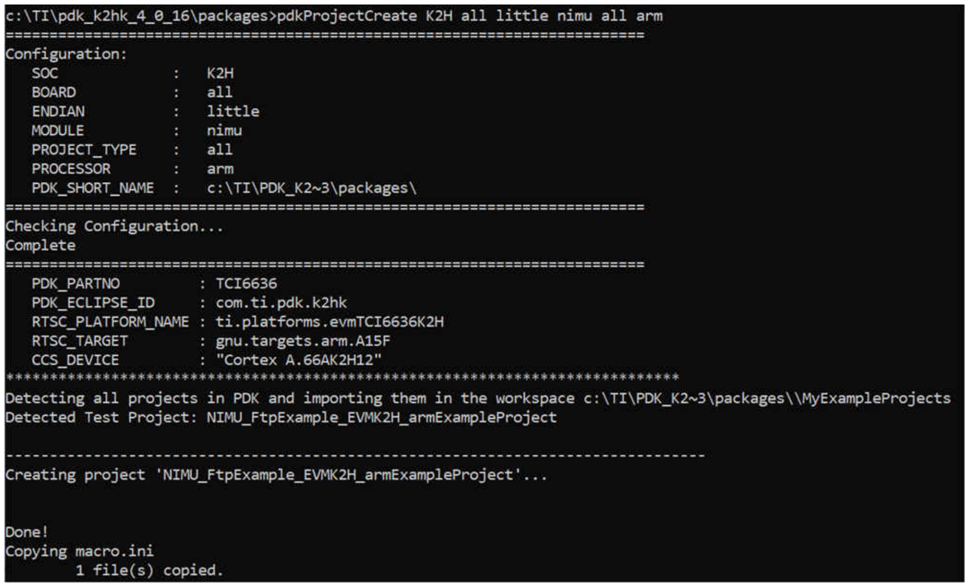
Keeping those differences in mind, one can create a FTP example project file for K2H, NIMU_INSTALL_DIR\example\ftpApp\k2h\armv7\bios\NIMU_FtpExample_EVMK2H_armExampleProject.txt, using NIMU_INSTALL_DIR\example\helloWorld\k2h\armv7\bios\NIMU_emacExample_EVMK2H_armBiosExampleProject.txt as the template. The text file shall have the following contents:

```
-ccs.linkFile "PDK_INSTALL_PATH/ti/transport/ndk/nimu/example/src/main_k2h.c"
-ccs.linkFile "PDK_INSTALL_PATH/ti/transport/ndk/nimu/example/ftpApp/ftpserver/ftp_commands.c"
-ccs.linkFile "PDK_INSTALL_PATH/ti/transport/ndk/nimu/example/ftpApp/ftpserver/ftp_filerout.c"
-ccs.linkFile "PDK_INSTALL_PATH/ti/transport/ndk/nimu/example/ftpApp/ftpserver/ftpserver.c"
-ccs.linkFile "PDK_INSTALL_PATH/ti/transport/ndk/nimu/example/helloWorld/src/nimu_osal.c"
-ccs.linkFile "PDK_INSTALL_PATH/ti/transport/ndk/nimu/example/helloWorld/src/
nimu_cppi_qmss_iface.c"
-ccs.linkFile "PDK_INSTALL_PATH/ti/transport/ndk/nimu/example/helloWorld/src/nimu_pa_iface.c"
-ccs.linkFile "PDK_INSTALL_PATH/ti/transport/ndk/nimu/example/helloWorld/src/setuprm.c"
-ccs.linkFile "PDK_INSTALL_PATH/ti/drv/rm/device/k2h/policy_dsp-only.c"
-ccs.linkFile "PDK_INSTALL_PATH/ti/drv/rm/device/k2h/policy_dsp_arm.c"
-ccs.linkFile "PDK_INSTALL_PATH/ti/drv/rm/device/k2h/global-resource-list.c"
-ccs.linkFile "PDK_INSTALL_PATH/ti/transport/ndk/nimu/example/helloWorld/k2h/armv7/bios/
helloWorld_ftp.cfg"
-ccs.setLinkerOptions " -lrdimon -lgcc -lm -lnosys -l:os.aa15fg -nostartfiles -static -Wl,--gc-
sections -L${NDK_INSTALL_DIR}/packages/ti/ndk/os/lib"
-ccs.setCompilerOptions "-c -mcpu=cortex-a15 -mtune=cortex-a15 -marm -mfloat-abi=hard -DSOC_K2H
-DDEVICE_K2H -DNIMU_FTP_APP -D_LITTLE_ENDIAN=1 -g -gstrict-dwarf -Wall -MMD -MP -I$
{PDK_INSTALL_PATH} -I{NDK_INSTALL_DIR}/packages"  -rtsc.enableRtsc
```

where:

- main_k2h.c is updated from the helloword.c, the major change is that it calls ftpserver_init() instead of DaemonNew()inside NetworkOpen()
- helloWorld_ftp.cfg is a duplicate copy of helloWorld.cfg for further modification as will be seen later

Then, the CCS project can be created by: `pdkProjectCreate K2H all little nimu all arm`.



```
c:\TI\pdk_k2hk_4_0_16\packages>pdkProjectCreate K2H all little nimu all arm
=========================================================================
Configuration:
   SOC              :   K2H
   BOARD            :   all
   ENDIAN           :   little
   MODULE           :   nimu
   PROJECT_TYPE     :   all
   PROCESSOR        :   arm
   PDK_SHORT_NAME   :   c:\TI\PDK_K2~3\packages\
=========================================================================
Checking Configuration...
Complete
=========================================================================
   PDK_PARTNO         : TCI6636
   PDK_ECLIPSE_ID     : com.ti.pdk.k2hk
   RTSC_PLATFORM_NAME : ti.platforms.evmTCI6636K2H
   RTSC_TARGET        : gnu.targets.arm.A15F
   CCS_DEVICE         : "Cortex A.66AK2H12"
*****************************************************************************
Detecting all projects in PDK and importing them in the workspace c:\TI\PDK_K2~3\packages\\MyExampleProjects
Detected Test Project: NIMU_FtpExample_EVMK2H_armExampleProject

-------------------------------------------------------------------------------
Creating project 'NIMU_FtpExample_EVMK2H_armExampleProject'...


Done!
Copying macro.ini
        1 file(s) copied.
```

Figure 3-3. K2H FTP Server Project Creation With Script

Finally, one can import the project into CCS and build:

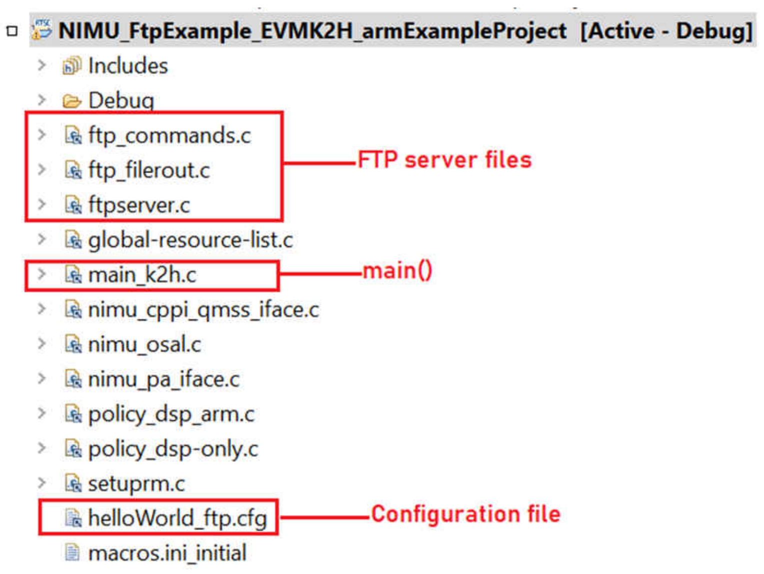

NIMU_FtpExample_EVMK2H_armExampleProject [Active - Debug]
- Includes
- Debug
- ftp_commands.c
- ftp_filerout.c
- ftpserver.c
- global-resource-list.c
- main_k2h.c
- nimu_cppi_qmss_iface.c
- nimu_osal.c
- nimu_pa_iface.c
- policy_dsp_arm.c
- policy_dsp-only.c
- setuprm.c
- helloWorld_ftp.cfg
- macros.ini_initial

Figure 3-4. K2H FTP Server CCS Project

Note

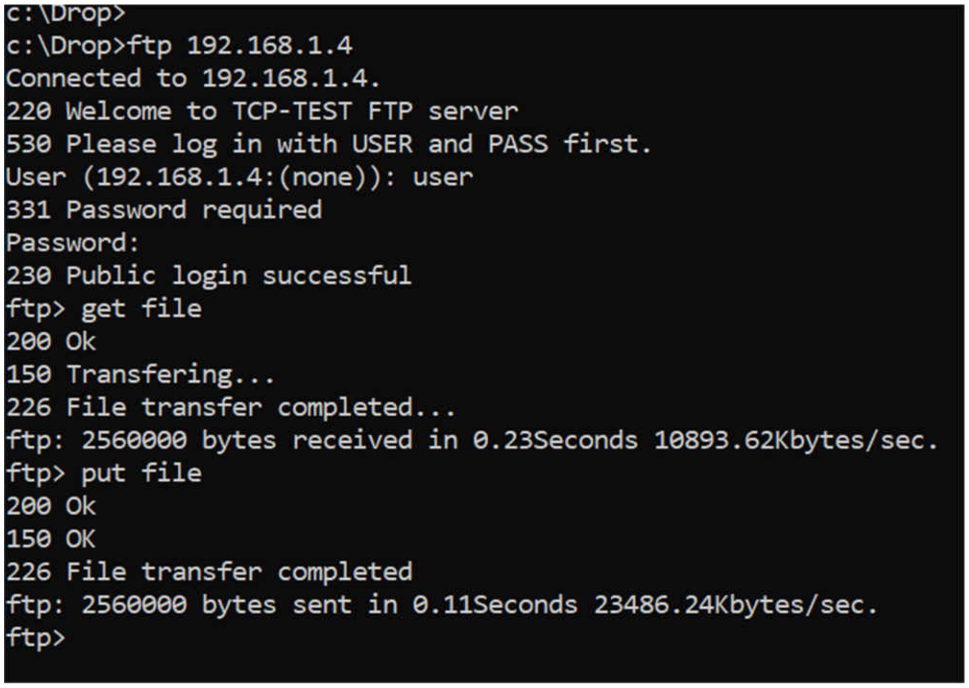
The FTP server source codes include header files from <ti/fs/fatfs/ff.h>. However the PDK K2H package doesn't contain them. One can copy those from PDK package for other devices like K2G or clone from TI GIT repository: . Three files are needed under pdk_k2hk_4_0_xx\packages\ti\fs\fatfs folder: ff.h, ffconf.h and integer.h.

3.3 Test K2H FTP Server Example

An out file called NIMU_FtpExample_EVMK2H_armExampleProject.out should be built. The example runs as the FTP server with a default IP address 192.168.1.4 on the K2H EVM. The PC acts as the FTP client with an IP address 192.168.1.11 as an example. The major test steps include:

- Set the K2H EVM in no-boot mode and use the standard GEL file for initialization
- Use CCS/JTAG to connect to the first A15 (arm_A15_0) core, load the program and run
- From the PC side, FTP to EVM with User: user and Password: password
- From the PC side, use get command to read a file from EVM into PC and put command to send a file from PC to EVM, corresponding to EVM transmission and reception, respectively

A sample test screenshot shown in Figure 3-5, showing the PC test commands and FTP throughput reported. This proves the functionality of the K2H FTP server example. Note that the server does not really host a file system on the K2H EVM, get/put commands are simply memory operations.



```
c:\Drop>
c:\Drop>ftp 192.168.1.4
Connected to 192.168.1.4.
220 Welcome to TCP-TEST FTP server
530 Please log in with USER and PASS first.
User (192.168.1.4:(none)): user
331 Password required
Password:
230 Public login successful
ftp> get file
200 Ok
150 Transfering...
226 File transfer completed...
ftp: 2560000 bytes received in 0.23Seconds 10893.62Kbytes/sec.
ftp> put file
200 Ok
150 OK
226 File transfer completed
ftp: 2560000 bytes sent in 0.11Seconds 23486.24Kbytes/sec.
ftp>
```

Figure 3-5. FTP Server Function Test With a PC

4 Performance Tuning

As seen from previous section, the EVM transmitting speed is approximately 11MB/s and receiving speed is approximately 23MB/s. There should be rooms for improvement. This section discusses methods and debugging techniques to improve the Ethernet throughput.

4.1 Quick Code Check

4.1.1 FTP Transmitting Code Check

Test result shows that 2,560,000 bytes data is transferred in a short duration. It is good to run a longer transfer with more data for measurement accuracy.

The transmitting is controlled inside ftp_filerout.c file:

```
int32_t ftp_filerout_read(io_handler_t *ioh, char *path)
{
    ……
    for (i=0; i<5000;i++)
    {
        send(ioh->data_socket, ioh->DataBuf, DATA_BUFFER_SIZE, 0);
        Task_yield();
    }
    .….
}
```

The DATA_BUFFER_SIZE is defined as 512. This is how 2,560,000 bytes (5,000 x 512) comes from. The packets captured on the wire show the packet size is 566 bytes, as a result of 54 bytes header (14 bytes Ethernet + 20 bytes IP + 20 bytes TCP) in addition to the 512 bytes data payload. This will not fully utilize the maximum Ethernet packet size which can be up to 1,514 bytes without fragmentation. One may update the DATA_BUFFER_SIZE to 1,460 (=1,514 – 54) and use a larger for() loop, e.g. 200,000 for test. This effectively transmits 292,000,000 bytes data with Maximum Transmission Unit (MTU) size.

Task_yield() is used to yield current task to other tasks with the same priority. It is unnecessary in such a simple test, so it can be commented out to tighten the for() loop.

4.1.2 FTP Receiving Code Check

One thing can be done to increase the throughput is to use the No-Copy APIs inside NDK. The regular socket APIs performs a CPU copy of the data received on the wire. On the contrary, the No-Copy APIs do not do that by merely passing the data pointer. You have to manually free the received buffer from APIs. For more information, see the *Enhanced No-Copy Socket Operation* section in the *TI Network Developer's Kit (NDK) API Reference Guide*.

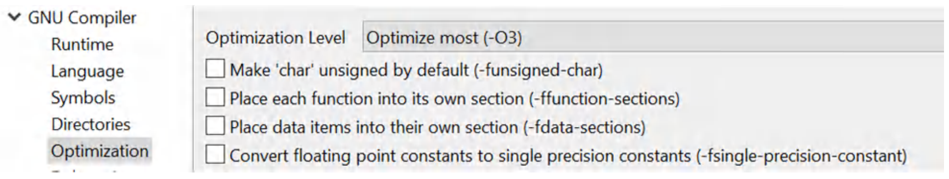
Checking the FTP receiving function in the same ftp_filerout.c file:

```
int32_t ftp_filerout_write(io_handler_t *ioh, char *path)
{
…
        bytesRead = (int)recvnc(ioh->data_socket, (void**)&pBuf, 0, &hBuffer);
…
}
```

The code already uses No-Copy API recvnc(), so no further change is needed.

4.1.3 CCS Project Optimization

The CCS project is created with –Og, which is optimized for debugging. This can be changed to –O3:



Optimization Level: Optimize most (-O3)
- [ ] Make 'char' unsigned by default (-funsigned-char)
- [ ] Place each function into its own section (-ffunction-sections)
- [ ] Place data items into their own section (-fdata-sections)
- [ ] Convert floating point constants to single precision constants (-fsingle-precision-constant)

Figure 4-1. Update FTP Server Example Compiler Optimization

Those are simple checks one can do very quickly, the throughput is improved to approximately 23MB/s for transmitting and receiving directions after the change.

4.2 Increase the TCP Buffer Sizes

The default TCP transmitting and receiving buffer sizes are defined in ndk_3_xx_xx_xx\packages\ti\ndk\inc\stack\inc\resif.h with 8192 bytes for devices with small memory footprint:

```
  #define DEF_SOCK_TCPTXBUF            8192
  #define DEF_SOCK_TCPRXBUF            8192
  #define DEF_SOCK_TCPRXLIMIT          8192
```

Those buffers can be increased in the FTP server main_k2h.c code before NC_NetStart() is called:

```
      //
      // This code sets up the TCP and UDP buffer sizes
      // (Note 8192 is actually the default. This code is here to
      // illustrate how the buffer and limit sizes are configured.)
      //
      // TCP Transmit buffer size
      rc = 65536;
      CfgAddEntry( hCfg, CFGTAG_IP, CFGITEM_IP_SOCKTCPTXBUF,
               CFG_ADDMODE_UNIQUE, sizeof(uint32_t), (uint8_t *)&rc, 0 );

      // TCP Receive buffer size (copy mode)
      CfgAddEntry( hCfg, CFGTAG_IP, CFGITEM_IP_SOCKTCPRXBUF,
               CFG_ADDMODE_UNIQUE, sizeof(uint32_t), (uint8_t *)&rc, 0 );

      // TCP Receive limit (non-copy mode)
      CfgAddEntry( hCfg, CFGTAG_IP, CFGITEM_IP_SOCKTCPRXLIMIT,
               CFG_ADDMODE_UNIQUE, sizeof(uint32_t), (uint8_t *)&rc, 0 );
```

65,536 bytes is the maximum size for a TCP buffer. However, with such changes, you may see the FTP connection is refused while ping to the EVM is still responsive.

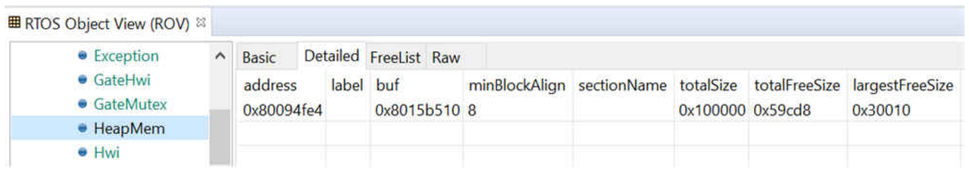
The most common reason is memory allocation failure. To check this, one can add print statement for failure code in socket() and accept() API calls inside ftpserver.c:

```
      if ((listen_sock = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP)) == INVALID_SOCKET)
      {
          printf("Failed to open listen socket with error %d\n", fdError());
  }

      if( (client_sock = accept(listen_sock, (struct sockaddr*)&client_addr, &len)) == INVALID_SOCKET)
      {
          /* Timeout, Wait a short time and try again */
       printf("Failed to accept with error %d\n", fdError());
          …
      }
```

As expected, the fdError() returns NDK_ENOMEM (12) in one of the error conditions. So the NDK memory needs to be increased. If another number returns, one can translate that number with ndk_3_xx_xx_xx\packages\ti\ndk\inc\serrno.h.

TCP buffers are directly allocated from heap, while all other NDK modules allocate buffers from NDK memory table, which is also allocated from heap. The heap size is controlled by BIOS.heapSize inside the .cfg file, increasing it from 0x40000 to 0x100000 resolves the FTP connection failure issue. The ROV view provides a snapshot of heap memory usage. The Ethernet throughput is improved to 50-60MB/s after increasing the TCP buffer sizes.



address	label	buf	minBlockAlign	sectionName	totalSize	totalFreeSize	largestFreeSize
0x80094fe4		0x8015b510	8		0x100000	0x59cd8	0x30010

Figure 4-2. Heap Memory RTOS Object View

4.3 UIA CPU Load Instrumentation

Before moving deeper into network setting and NIMU driver tuning, it is good to understand if the throughput is limited by the CPU load, which is probably 100% busy? Such can be answered by using Unified Instrumentation Architecture (UIA) module for CPU load measurement. The major steps include:

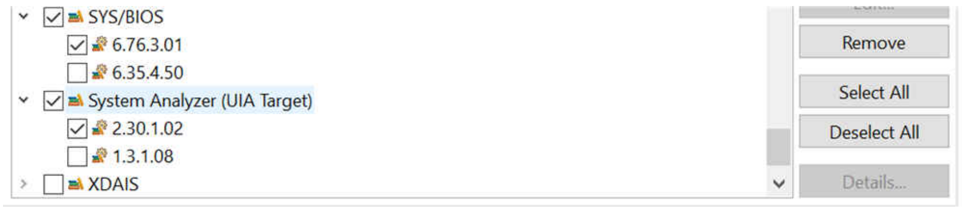


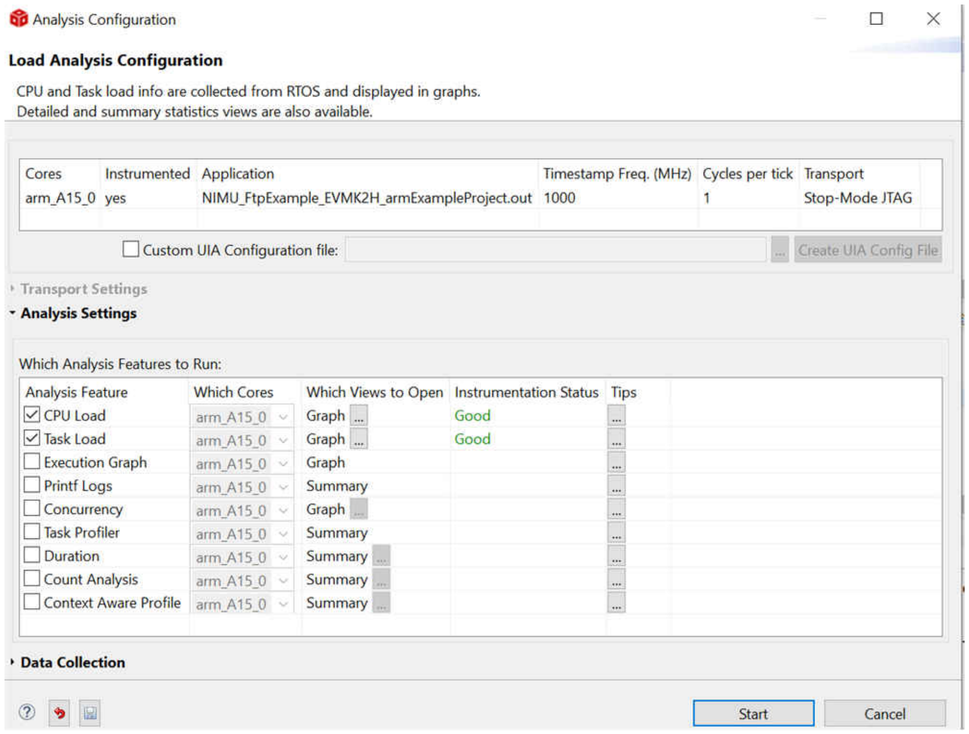
Figure 4-3. Adding UIA Module Into CCS Project

1. Add the below configuration into the .cfg file. The SYSBIOS has a default CPU speed info which needs to be updated with the real CPU speed as well:

```
BIOS.cpuFreq.lo = 1000000000;  /*1GHz CPU*/

var LoggingSetup = xdc.useModule('ti.uia.sysbios.LoggingSetup');
LoggingSetup.loadLogging = true;
LoggingSetup.loadLoggerSize = 32768;
LoggingSetup.mainLoggerSize = 32768;
LoggingSetup.sysbiosLoggerSize = 32768;
LoggingSetup.sysbiosSemaphoreLogging = true;
LoggingSetup.loadTaskLogging = true;
LoggingSetup.loadSwiLogging = true;
LoggingSetup.loadHwiLogging = true;
LoggingSetup.enableTaskProfiler = true;
LoggingSetup.eventUploadMode = LoggingSetup.UploadMode_JTAGSTOPMODE;
```

2. Enable the UIA measurement in CCS by selecting Tools → RTOS Analyzer → Load Analysis:



Load Analysis Configuration

CPU and Task load info are collected from RTOS and displayed in graphs.
Detailed and summary statistics views are also available.

Cores	Instrumented	Application	Timestamp Freq. (MHz)	Cycles per tick	Transport
arm_A15_0	yes	NIMU_FtpExample_EVMK2H_armExampleProject.out	1000	1	Stop-Mode JTAG

☐ Custom UIA Configuration file: Create UIA Config File

Which Analysis Features to Run:

Analysis Feature	Which Cores	Which Views to Open	Instrumentation Status	Tips
☑ CPU Load	arm_A15_0	Graph	Good	...
☑ Task Load	arm_A15_0	Graph	Good	...
☐ Execution Graph	arm_A15_0	Graph		...
☐ Printf Logs	arm_A15_0	Summary		...
☐ Concurrency	arm_A15_0	Graph		...
☐ Task Profiler	arm_A15_0	Summary		...
☐ Duration	arm_A15_0	Summary		...
☐ Count Analysis	arm_A15_0	Summary		...
☐ Context Aware Profile	arm_A15_0	Summary		...

Figure 4-4. UIA Analysis Configuration

3. Run the test.
4. Halt CPU to collect the CPU load graph. Below is an example with Tx throughput 61MB/s followed by Rx throughput 59MB/s. The CPU load is 55% and 68%, respectively, so there is still CPU processing power left.

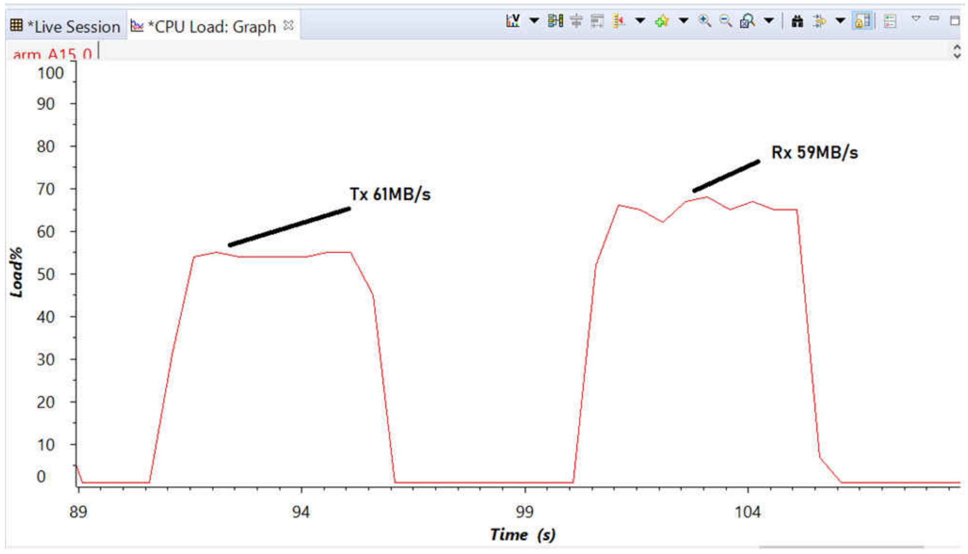


Figure 4-5. A15 CPU Load Under FTP Transmitting and Receiving

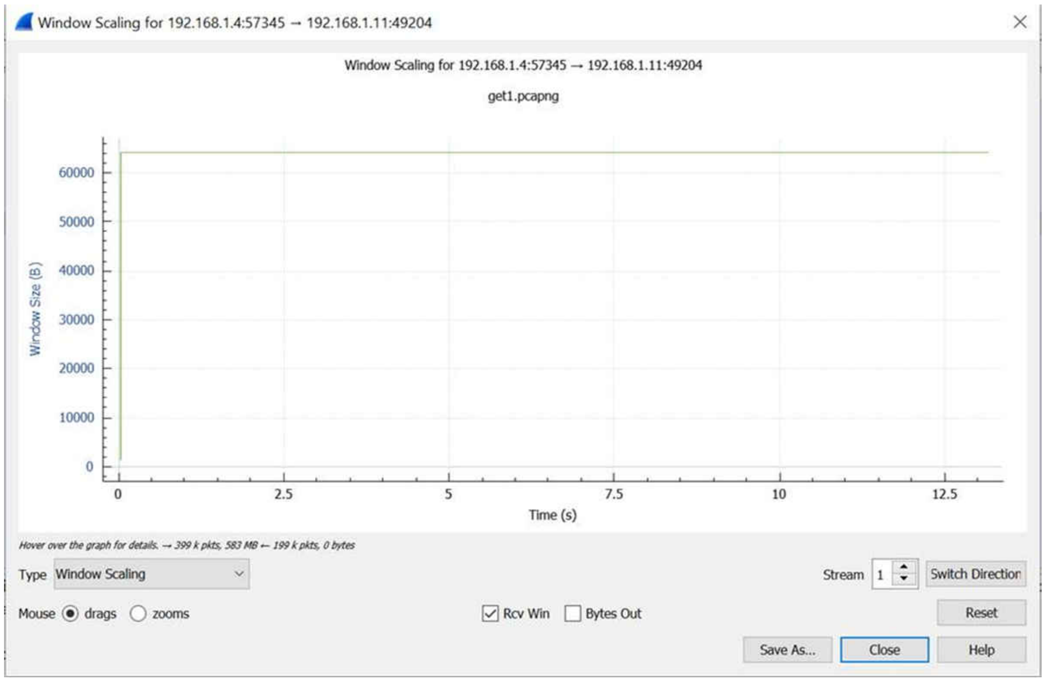
5. Note that the UIA module adds a little bit overhead as every context switch is being timestamped to get more accurate CPU load number. Remove the UIA code in the .cfg file once the measurement is performed.

4.4 What Can We Do on the PC Side?

Wireshark is a good tool to capture and analyze packet traffic for further insight. A usage example is illustrated below for a long data transfer from the K2H EVM to PC with throughput 45MB/s.

4.4.1 TCP Window Scaling Check

Through TCP stream analysis, it shows the TCP window size is always 64KB, no scaling happened.



Hover over the graph for details. → 399 k pkts, 583 MB ← 199 k pkts, 0 bytes

Type: Window Scaling

Mouse: [✓ drags] zooms

[✓ Rcv Win] Bytes Out

Figure 4-6. TCP Window Size

The stream analysis also shows Round Trip Time (RTT) is about 0.60 millisecond mostly:

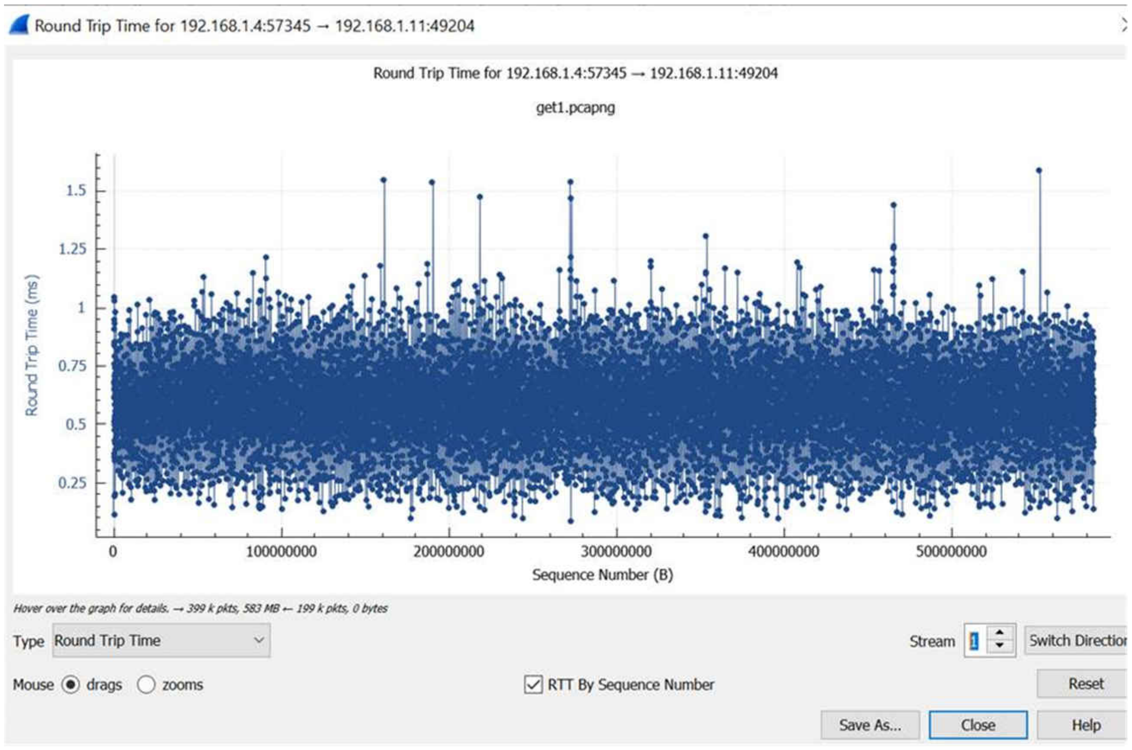
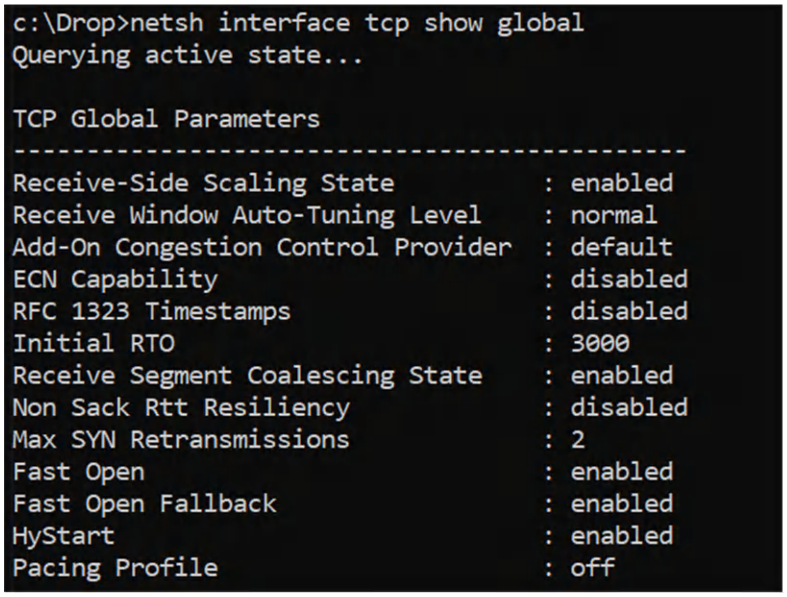


Figure 4-7. TCP Round Trip Time

With this RTT and TCP window size, the transfer supports: 64KB/0.6 ms = 107MB/s throughput. The TCP window size is big enough even without scaling, this is not a factor limiting the throughput. Nevertheless, it is worthy to check and make sure the TCP window scaling is enabled on the Windows PC side, by using command: netsh interface tcp show global. A larger TCP window helps to improve the throughput especially in high RTT scenario.



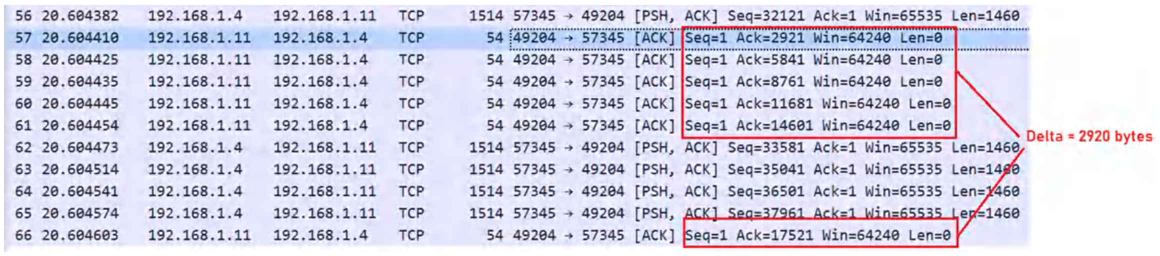
```
c:\Drop>netsh interface tcp show global
Querying active state...

TCP Global Parameters
----------------------------------------------
Receive-Side Scaling State          : enabled
Receive Window Auto-Tuning Level    : normal
Add-On Congestion Control Provider  : default
ECN Capability                      : disabled
RFC 1323 Timestamps                 : disabled
Initial RTO                         : 3000
Receive Segment Coalescing State    : enabled
Non Sack Rtt Resiliency             : disabled
Max SYN Retransmissions             : 2
Fast Open                           : enabled
Fast Open Fallback                  : enabled
HyStart                             : enabled
Pacing Profile                      : off
```

Figure 4-8. TCP Scaling Check on Windows PC

4.4.2 Receive Interrupt Coalescing Check

As shown from the packet capture, the data transfer size is 1514 bytes from the K2H EVM to the PC. The PC sends an ACK packet for every 2920 bytes (2 packets) received by examining the successive ACK packets' acknowledge number field:

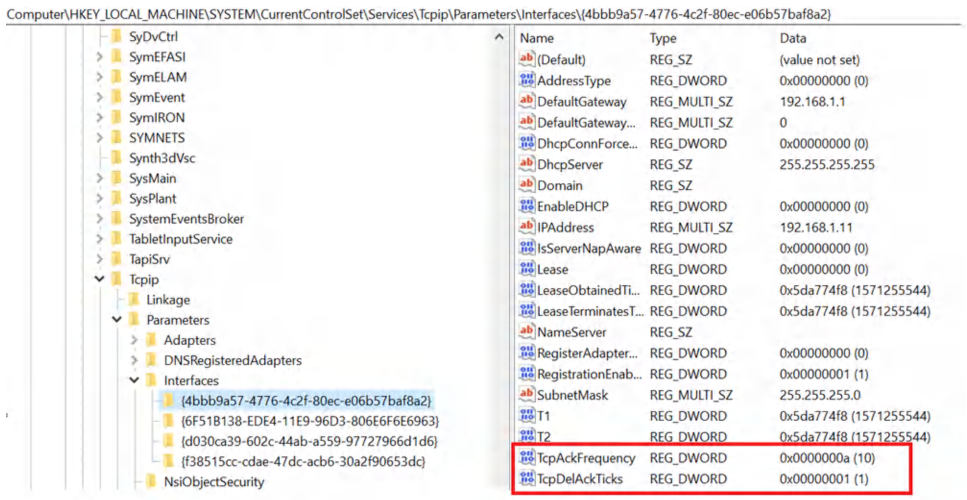


```
56 20.604382   192.168.1.4    192.168.1.11   TCP     1514 57345 → 49204 [PSH, ACK] Seq=32121 Ack=1 Win=65535 Len=1460
57 20.604410   192.168.1.11   192.168.1.4    TCP       54 49204 → 57345 [ACK] Seq=1 Ack=2921 Win=64240 Len=0
58 20.604425   192.168.1.11   192.168.1.4    TCP       54 49204 → 57345 [ACK] Seq=1 Ack=5841 Win=64240 Len=0
59 20.604435   192.168.1.11   192.168.1.4    TCP       54 49204 → 57345 [ACK] Seq=1 Ack=8761 Win=64240 Len=0
60 20.604445   192.168.1.11   192.168.1.4    TCP       54 49204 → 57345 [ACK] Seq=1 Ack=11681 Win=64240 Len=0
61 20.604454   192.168.1.11   192.168.1.4    TCP       54 49204 → 57345 [ACK] Seq=1 Ack=14601 Win=64240 Len=0
62 20.604473   192.168.1.4    192.168.1.11   TCP     1514 57345 → 49204 [PSH, ACK] Seq=33581 Ack=1 Win=65535 Len=1460
63 20.604514   192.168.1.4    192.168.1.11   TCP     1514 57345 → 49204 [PSH, ACK] Seq=35041 Ack=1 Win=65535 Len=1460
64 20.604541   192.168.1.4    192.168.1.11   TCP     1514 57345 → 49204 [PSH, ACK] Seq=36501 Ack=1 Win=65535 Len=1460
65 20.604574   192.168.1.4    192.168.1.11   TCP     1514 57345 → 49204 [PSH, ACK] Seq=37961 Ack=1 Win=65535 Len=1460
66 20.604603   192.168.1.11   192.168.1.4    TCP       54 49204 → 57345 [ACK] Seq=1 Ack=17521 Win=64240 Len=0
```

Figure 4-9. Wireshark Analysis of ACK Packet Frequency

The frequent reception of TCP ACK packets drives up the CPU usage during receiving interrupt handling. The way to reduce the interrupt generation and hence the interrupt handling is called interrupt coalescing. It prevents interrupts from being raised to a CPU until a specific amount of events are pending. The CPU is relaxed and the throughput can be potentially increased with fewer interrupts.

Interrupt coalescing can be explored on either side, or both sides. From the Windows 10 PC side, if the generated TCP ACK packets can be suppressed to less frequent, then the K2H CPU saves the power for receiving interrupt handling. The interrupt coalescing is configured in the Windows 10 registry under Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters\Interfaces\{Interface_name}, one can use regedit utility to add two additional fields: TcpAckFrequency and TcpDelAckTicks. An ACK packet is transmitted every receiving number of TcpAckFrequency packets or when TcpDelAckTicks (in unit of 100 millisecond) delay timer expires, whichever happens first. An example is depicted below:



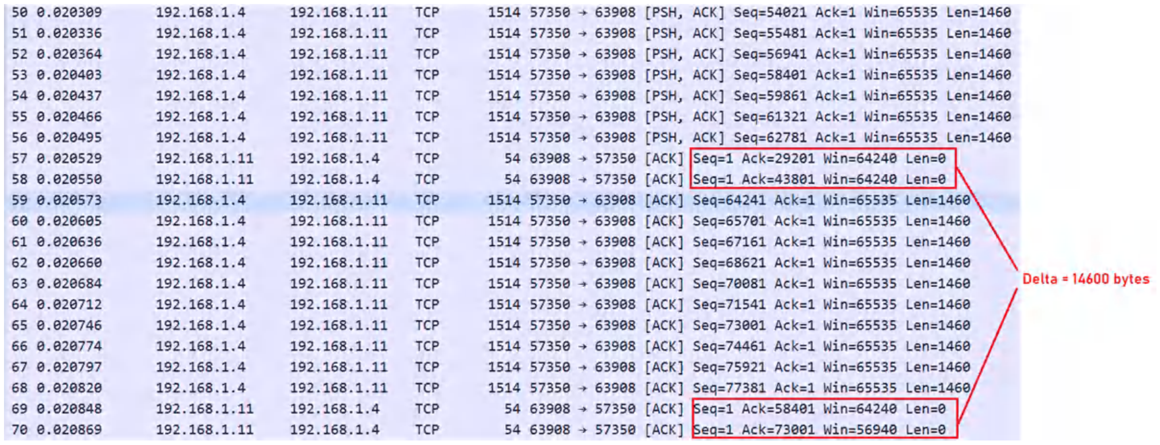
Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters\Interfaces\{4bbb9a57-4776-4c2f-80ec-e06b57baf8a2}

```
SyDvCtrl
SymEFASI
SymELAM
SymEvent
SymIRON
SYMNETS
Synth3dVsc
SysMain
SysPlant
SystemEventsBroker
TabletInputService
TapiSrv
Tcpip
    Linkage
    Parameters
        Adapters
        DNSRegisteredAdapters
        Interfaces
            {4bbb9a57-4776-4c2f-80ec-e06b57baf8a2}
            {6F51B138-EDE4-11E9-96D3-806E6F6E6963}
            {d030ca39-602c-44ab-a559-97727966d1d6}
            {f38515cc-cdae-47dc-acb6-30a2f90653dc}
        NsiObjectSecurity
```

Name	Type	Data
(Default)	REG_SZ	(value not set)
AddressType	REG_DWORD	0x00000000 (0)
DefaultGateway	REG_MULTI_SZ	192.168.1.1
DefaultGateway...	REG_MULTI_SZ	0
DhcpConnForce...	REG_DWORD	0x00000000 (0)
DhcpServer	REG_SZ	255.255.255.255
Domain	REG_SZ	
EnableDHCP	REG_DWORD	0x00000000 (0)
IPAddress	REG_MULTI_SZ	192.168.1.11
IsServerNapAware	REG_DWORD	0x00000000 (0)
Lease	REG_DWORD	0x00000000 (0)
LeaseObtainedTi...	REG_DWORD	0x5da774f8 (1571255544)
LeaseTerminatesT...	REG_DWORD	0x5da774f8 (1571255544)
NameServer	REG_SZ	
RegisterAdapter...	REG_DWORD	0x00000000 (0)
RegistrationEnab...	REG_DWORD	0x00000001 (1)
SubnetMask	REG_MULTI_SZ	255.255.255.0
T1	REG_DWORD	0x5da774f8 (1571255544)
T2	REG_DWORD	0x5da774f8 (1571255544)
TcpAckFrequency	REG_DWORD	0x0000000a (10)
TcpDelAckTicks	REG_DWORD	0x00000001 (1)

Figure 4-10. Windows PC Interrupt Coalescing Configuration

Those numbers are based on trials and the PC needs to reboot for the setting to take into effect. With TcpAckFrequency = 10 and TcpDelAckTicks = 1, the ACK is now transmitted for every 10 (from 2) received packets:



```
50 0.020309    192.168.1.4    192.168.1.11   TCP    1514 57350 → 63908 [PSH, ACK] Seq=54021 Ack=1 Win=65535 Len=1460
51 0.020336    192.168.1.4    192.168.1.11   TCP    1514 57350 → 63908 [PSH, ACK] Seq=55481 Ack=1 Win=65535 Len=1460
52 0.020364    192.168.1.4    192.168.1.11   TCP    1514 57350 → 63908 [PSH, ACK] Seq=56941 Ack=1 Win=65535 Len=1460
53 0.020403    192.168.1.4    192.168.1.11   TCP    1514 57350 → 63908 [PSH, ACK] Seq=58401 Ack=1 Win=65535 Len=1460
54 0.020437    192.168.1.4    192.168.1.11   TCP    1514 57350 → 63908 [PSH, ACK] Seq=59861 Ack=1 Win=65535 Len=1460
55 0.020466    192.168.1.4    192.168.1.11   TCP    1514 57350 → 63908 [PSH, ACK] Seq=61321 Ack=1 Win=65535 Len=1460
56 0.020495    192.168.1.4    192.168.1.11   TCP    1514 57350 → 63908 [PSH, ACK] Seq=62781 Ack=1 Win=65535 Len=1460
57 0.020529    192.168.1.11   192.168.1.4    TCP      54 63908 → 57350 [ACK] Seq=1 Ack=29201 Win=64240 Len=0
58 0.020550    192.168.1.11   192.168.1.4    TCP      54 63908 → 57350 [ACK] Seq=1 Ack=43801 Win=64240 Len=0
59 0.020573    192.168.1.4    192.168.1.11   TCP    1514 57350 → 63908 [ACK] Seq=64241 Ack=1 Win=65535 Len=1460
60 0.020605    192.168.1.4    192.168.1.11   TCP    1514 57350 → 63908 [ACK] Seq=65701 Ack=1 Win=65535 Len=1460
61 0.020636    192.168.1.4    192.168.1.11   TCP    1514 57350 → 63908 [ACK] Seq=67161 Ack=1 Win=65535 Len=1460
62 0.020660    192.168.1.4    192.168.1.11   TCP    1514 57350 → 63908 [ACK] Seq=68621 Ack=1 Win=65535 Len=1460
63 0.020684    192.168.1.4    192.168.1.11   TCP    1514 57350 → 63908 [ACK] Seq=70081 Ack=1 Win=65535 Len=1460
64 0.020712    192.168.1.4    192.168.1.11   TCP    1514 57350 → 63908 [ACK] Seq=71541 Ack=1 Win=65535 Len=1460
65 0.020746    192.168.1.4    192.168.1.11   TCP    1514 57350 → 63908 [ACK] Seq=73001 Ack=1 Win=65535 Len=1460
66 0.020774    192.168.1.4    192.168.1.11   TCP    1514 57350 → 63908 [ACK] Seq=74461 Ack=1 Win=65535 Len=1460
67 0.020797    192.168.1.4    192.168.1.11   TCP    1514 57350 → 63908 [ACK] Seq=75921 Ack=1 Win=65535 Len=1460
68 0.020820    192.168.1.4    192.168.1.11   TCP    1514 57350 → 63908 [ACK] Seq=77381 Ack=1 Win=65535 Len=1460
69 0.020848    192.168.1.11   192.168.1.4    TCP      54 63908 → 57350 [ACK] Seq=1 Ack=58401 Win=64240 Len=0
70 0.020869    192.168.1.11   192.168.1.4    TCP      54 63908 → 57350 [ACK] Seq=1 Ack=73001 Win=56940 Len=0
```

Figure 4-11. Effect of Interrupt Coalescing Setting on ACK Packet Generation

4.5 What Else Can We Do on the K2H Side?

4.5.1 TCP/IP Checksum Offloading Check

In the K2H device, Packet Accelerator (PA) is one of the main components of the network coprocessor peripheral. The PA is a firmware that works with the Gigabit Ethernet switch subsystem to perform packet processing operations such as packet header classification, checksum generation, and multi-queue routing. On the other hand, the NDK software supports both Layer 3 and Layer 4 checksums calculation. If this can be offloaded to the K2H's PA, then the CPU processing power can be saved.

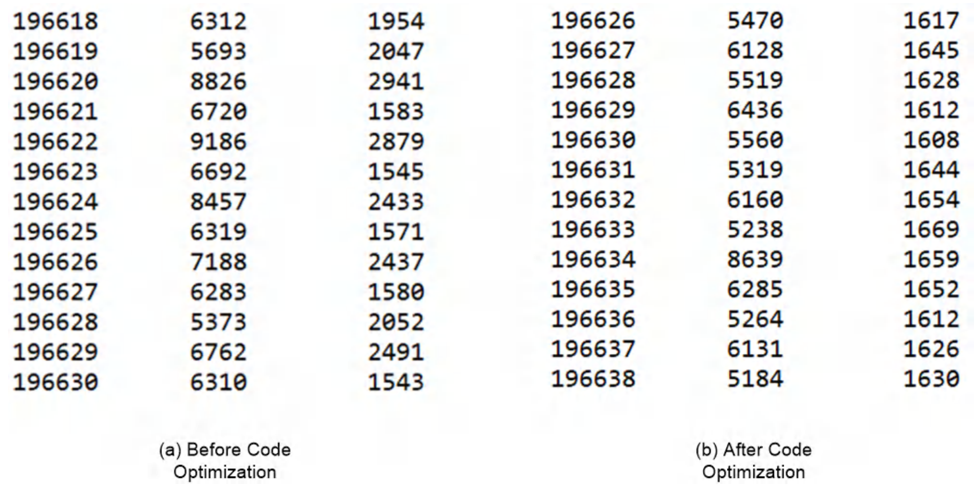
Current NDK design abstracts at NIMU layer. NDK is device independent without the knowledge of underlying hardware capabilities. In the receiving path, the PA does the packet classification at Layer 2 then handles the packets to the NDK stack. In the transmitting path, the NDK forms the packets and pass them to the NIMU driver. All the Layer 3 and Layer 4 are handled by the NDK software, the checksum calculation can't be offloaded to the PA firmware.

4.5.2 NIMU Driver Efficiency Profiling

The NIMU driver source code is NIMU_INSTALL_DIR\src\v2\nimu_eth.c. The transmitting function is EmacSend() and the receiving function is EmacRxPktISR(). There is a debug macro TIMING can be enabled to profile the CPU cycles spent in transmitting and receiving routines. The timestamp function is nimuUtilReadTime32(). One may further expand this debug to record additional cycle count information for each part of the code, and judge where efforts probably can be spent for driver code optimization.

For example, the transmitting packet number counter, the delta from last time EmacSend() is entered and total time spent in the EmacSend() are saved in a circular debug buffer for further analysis, as shown below. The EmacSend() function is found to take about 1550 to 2950 cycles (the third column).

By looking into the implementation of EmacSend(), the driver does transmitting return queue processing (in a loop over number of descriptors in gTxReturnQHnd), it pops the gTxReturnQHnd and it returns PBM packet back to the NDK stack, and pushes transmitting descriptor to the transmitting free queue, in addition to actually sending the packet out. After some code optimization, the cycle is reduced to about 1600 in the same routine, saving up to 45%.



196618	6312	1954
196619	5693	2047
196620	8826	2941
196621	6720	1583
196622	9186	2879
196623	6692	1545
196624	8457	2433
196625	6319	1571
196626	7188	2437
196627	6283	1580
196628	5373	2052
196629	6762	2491
196630	6310	1543

(a) Before Code Optimization

196626	5470	1617
196627	6128	1645
196628	5519	1628
196629	6436	1612
196630	5560	1608
196631	5319	1644
196632	6160	1654
196633	5238	1669
196634	8639	1659
196635	6285	1652
196636	5264	1612
196637	6131	1626
196638	5184	1630

(b) After Code Optimization

Figure 4-12. Code Cycle Count Analysis Before and After Code Optimization

Note that the NIMU library needs to be rebuilt and linked with the application for the changes taking into effect.

4.5.3 Receive Interrupt Coalescing

The idea of interrupt coalescing is experimented on the PC side in Section 4.4.2. This can also be achieved by letting the ACK packets accumulated to certain levels for one-time processing, in case that PC is unable to throttle the ACK packet generation.

This feature is supported by the K2H and explained in the (*KeyStone Architecture Multicore Navigator User's Guide*) under field *Interrupt Pacing Mode* with 4 options:

- 0 = None — interrupt on entry threshold count only (list full)
- 1 = Time delay since last interrupt. This produces a periodic interrupt (as long as the list does not fill early and descriptors continue arriving).
- 2 = Time delay since first new packet. This starts the timer countdown with the first packet received following a previous interrupt.
- 3 = Time delay since last new packet. This restarts the timer countdown with each new packet.

The interrupt pacing mode works with another parameter timerLoadCount, which counts down a global timer ticks to delay interrupt. The pacing mode is configured in the setup_rx_queue() of driver code nimu_eth.c:

```
accCfg.timerLoadCount      =   0;
accCfg.interruptPacingMode =   Qmss_AccPacingMode_LAST_INTERRUPT;
```

After some trials with different settings and cross-checking with Processor SDK Linux K2H usage in:

- – accumulator = <0 47 16 2 50>
- and knav_init_acc_range() in https://git.ti.com/cgit/processor-sdk/processor-sdk-linux/tree/drivers/soc/ti/knav_qmss_acc.c?h=processor-sdk-linux-4.19.y:

```
The NIMU driver code is updated with the following:

    accCfg.timerLoadCount      =   2;    /* 50us/ACC_DEFAULT_PERIOD(25us by firmware) = 2 */
    accCfg.interruptPacingMode =   Qmss_AccPacingMode_FIRST_NEW_PACKET;
```

4.6 Final FTP Throughput Results

With all the changes integrated, the K2H FTP server example achieves transmitting throughput 60-73MB/s and receiving throughput 67MB/s, both well above the initial results.

5 Summary

This application report explains how to create the FTP server application on TI K2H device, using TI processor SDK and NDK software offerings with RTOS. The example runs on an Arm A15 core and its network throughput performance is measured. The steps creating the Arm CCS project also apply to DSP CCS project. The same can be extended to other TI devices supported by Processor SDK RTOS, which share the same NIMU driver and NDK stack architecture. In addition, the document illustrates how to measure CPU load using TI UIA tool, to understand if the CPU processing power a limitation factor.

The document largely explores the network performance tuning by various approaches. At first, application level code check and project build optimization are tried for simple throughput boost. Next, the NDK driver is updated with larger TCP buffer sizes. A typical memory allocation issue arises and it is dealt with. Moreover, PC side improvement is studied by analyzing the packet traffic and implementing interrupt coalescing from TCP ACK packet generation perspective. Finally, the K2H NIMU driver is optimized and adjusted for receiving packet interrupt coalescing as an advanced topic.

While the K2H NIMU driver and NDK are TI devices specific, the topics such as TCP window scaling, checksum offloading and interrupt coalescing are not limited to this and they are applicable to generic TCP/IP network performance tuning. Collectively with all the changes, significant throughput improvement is achieved. Certainly there can be more works, like implementing jumbo packets, ensuring the TCP window size is scaled up, offloading checksum calculation to hardware by redesigning the NDK, NIMU interfaces and so forth. Those may be investigated in the future.

6 References

- Texas Instruments: *TI Network Developer's Kit (NDK) API Reference Guide*
- Texas Instruments: *(KeyStone Architecture Multicore Navigator User's Guide*