Application Report
SPRAAX4 – July 2008

Getting Started With the C6000 Network Development Kit (NDK)

Eric Wilbur — *Technical Training Organization*
Rafael de Souza — *Software Development Organization*
Mohsen Khayami — *Digital Field Applications*

ABSTRACT

This guide was written for beginning and intermediate users of the C6000 Network Development Kit (NDK). After downloading the evaluation version or purchasing the NDK, new users should read this guide and do the corresponding labs to gain familiarity with the NDK directories, files, and working code examples. Most of the "getting started" challenges which new users face when they first open the box are covered in detail.

Project collateral and source code discussed in this document can be downloaded from the following URL: http://www-s.ti.com/sc/techlit/spraax4.zip.

Contents

1 Introduction 4
2 Introduction to Networking 5
2.1 Client-Server Paradigm 5
2.2 Networking Overview 6
2.3 OSI Model 7
2.4 TCP/IP Implentation of the OSI Model 8
2.5 Networking – Terms and Definitions 9
2.6 Anatomy of an Ethernet Frame 10
3 EMAC Architecture Overview 11
3.1 EMAC is a Master Peripheral 11
3.2 Media Independent Interfaces (MII) 12
3.3 EMAC Module 13
3.4 EMAC Data Layer – Example (TX) 14
4 Network Development Kit (NDK) 15
4.1 TI's NDK Implementation 15
4.2 Network Development Kit (NDK) Contents 16
4.3 Purchasing the NDK 17
4.4 Installing the NDK (some highlights) 18
5 Client.pjt – Your "StarterWare" 19
5.1 Client.pjt Example – Overview 19
5.2 Client.pjt – What Does it Contain? 20
5.3 Client.pjt – Libraries 21

6 Lab 1: Code Composer Studio (CCS) Setup 22
6.1 Lab Overview: 22
6.2 Hardware Setup 23
6.3 CCS Setup 23
6.4 Set up CCS – Customize Options 25
7 Lab 2: Analyze Audio Pass Thru Application 28
7.1 Lab Overview: 28
7.2 Open the Project, Build/Run 29
7.3 Debugging Techniques 30
7.4 Analyze TCF (Text Configuration File) 31
8 Lab 3: Using Client.pjt "Out of the Box" 33
8.1 Lab Overview: 33
8.2 Open Client.pjt and Run It 34
8.3 Play With the NDK Services in Client.pjt 37
9 Lab 4 – Combining the NDK and Audio Lab 40
9.1 Lab Overview: 40
9.2 Introduction 41
9.3 Modify Client.pjt to Include Application Files/Libraries 41
9.4 Modify Source Files 43
9.5 Port DSP/BIOS Settings From Application to NDK 46
9.6 Build, Load and RUN ! 47
10 Modifying Client.pjt 48
10.1 Deleting Client.pjt Source Files 48
10.2 Client.pjt – Modifying Newservers.c 49
10.3 Client.c Source File – Overview 50
10.4 Client.c – Remove Telnet Service 50
10.5 Client.c – Remove HTTP Service 51
10.6 Client.c – Remove "USE_OLD_SERVERS" 51
10.7 Client.c – Remove Unused DAEMON Servers 52
10.8 Client.c – Clean Up 52
11 Lab 5 – Modify Client.pjt 53
11.1 Lab Overview: 53
11.2 Modify Client.pjt 54
11.3 Save Your Solution and Close CCS 62
12 Sockets Programming 63
12.1 What is a Socket? 63
12.2 TCP – Sockets Programming APIs 64
12.3 TCP – Application Example (echosrv.c) 65
12.4 File Descriptor (FD) Environment 66
12.5 Echosrv.c – Deep Dive (1/4) 67
12.6 Echosrv.c – Deep Dive (2/4) 68
12.7 Echosrv.c – Deep Dive (3/4) 69
12.8 Echosrv.c – Deep Dive (4/4) 70
12.9 UDP – Sockets Programming APIs 71
12.10 TCP vs. UDP 72
12.11 Sockets Programming vs. DAEMON 73
13 Lab 6 – Use Sockets Programming APIs 74
13.1 Lab Overview: 74
13.2 Use Sockets Programming to SEND Data 75

13.3 Save Your Solution and Close CCS 77
14 Conclusion 78
14.1 NDK Considerations 78
14.2 Update – New NDK version 1.94 79
14.3 For More Information… 79
15 Advanced (but useful) Topics 81
15.1 Introduction 81
15.2 Callback Functions 82
15.3 Order of Events (inside the NDK…) 82

1 Introduction

This guide was written for beginning and intermediate users of the C6000 Network Development Kit (NDK). After downloading the evaluation version or purchasing the NDK, new users should read this guide and do the corresponding labs to gain familiarity with the NDK directories, files, and working code examples. Most of the "getting started" challenges which new users face when they first open the box are covered in detail. The intent of this guide is NOT to replace the NDK User Guide (SPRU523) and the NDK Programmers Guide (SPRU524) – but to complement the content of these documents and increase the user's out-of-box experience.

This guide may contain information you already know. The goal of this guide was to be comprehensive so that users with different experience levels could either read the guide from start to finish (which is recommended) or locate the specific information that is required and only read/study those sections.

There are 6 labs contained in this guide. The first two are simply getting used to Code Composer Studio (CCS) and a simple audio pass-through lab that shows basic DSP/BIOS threads. The 3rd lab shows you how to open up one of the key NDK examples (client.pjt) and run it. This is the first introduction to the NDK software. The 4th lab walks you through combining the NDK and the audio application – this is by far the most critical lab. The 5th lab shows you step-by-step how to modify client.pjt to do one specific thing – a DAEMON server. This is a helpful example regarding how to modify client.pjt to do specifically what you need in your application. Lab 6 walks you through a sockets programming example.

After successfully reading this guide and doing the labs, the next step is to read chapters 1-3 of the NDK User Guide (SPRU523). This document contains many useful tips that were not duplicated in this Getting Started Guide. If you plan to use sockets programming APIs, the NDK Programmers Guide (SPRU524) is a must read because it contains details regarding all of the supported sockets programming APIs.

The NDK is a software layer (actually, several layers) that uses TI's real-time kernel, DSP/BIOS, which configures, controls and uses the hardware EMAC (Ethernet Media Access Controller) peripheral. There are two types of users of the EMAC: (1) users who know sockets programming and simply want to use the NDK as is; (2) users who are doing their own software layer above the EMAC and need to know the details of the EMAC architecture. This getting started guide is focused on the NDK software and therefore will not go into much detail on the EMAC hardware architecture. For more information about the EMAC peripheral, refer to SPRU975 – EMAC User Guide.

NOTE: While this guide and labs are based on the TMS320C6455 device and DSP Starter Kit (DSK), the concepts presented here can be applied to any NDK user on any platform. You will simply need to pay close attention to the specific path names, file names and/or directories based upon the platform being used. The NDK code is the same for any platform (such as DM648, DM6437/C6424) – only the HAL (hardware abstraction layer) libraries are specific to the development board being used.

2 Introduction to Networking

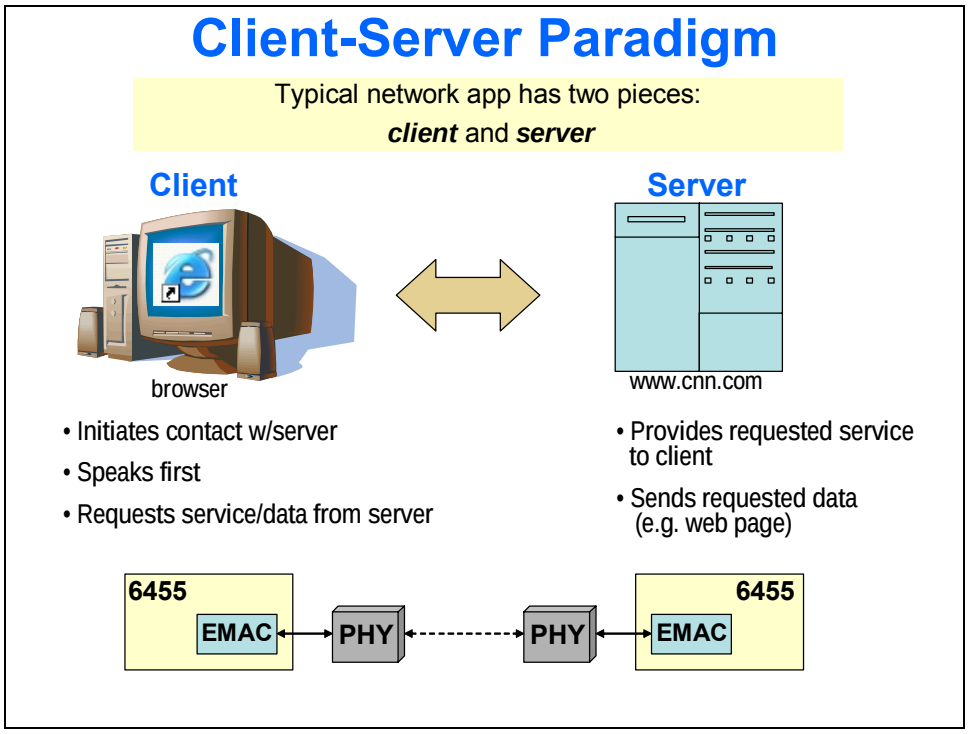
2.1 Client-Server Paradigm

Networking is basically a set of processing nodes used for data communication. From a high level, you have client and a server. The client requests information from a server and the server responds by sending the client the requested data. When you open a browser on your computer and type in a URL, the server (e.g. www.cnn.com) magically appears on your browser screen. What happens in that relatively small amount of time (from request to response) can be quite complex.

However, standards have helped make this type of communication relatively straightforward. Fairly rigid protocols have been developed to help both client and server application programmers get their networking system working rather quickly.

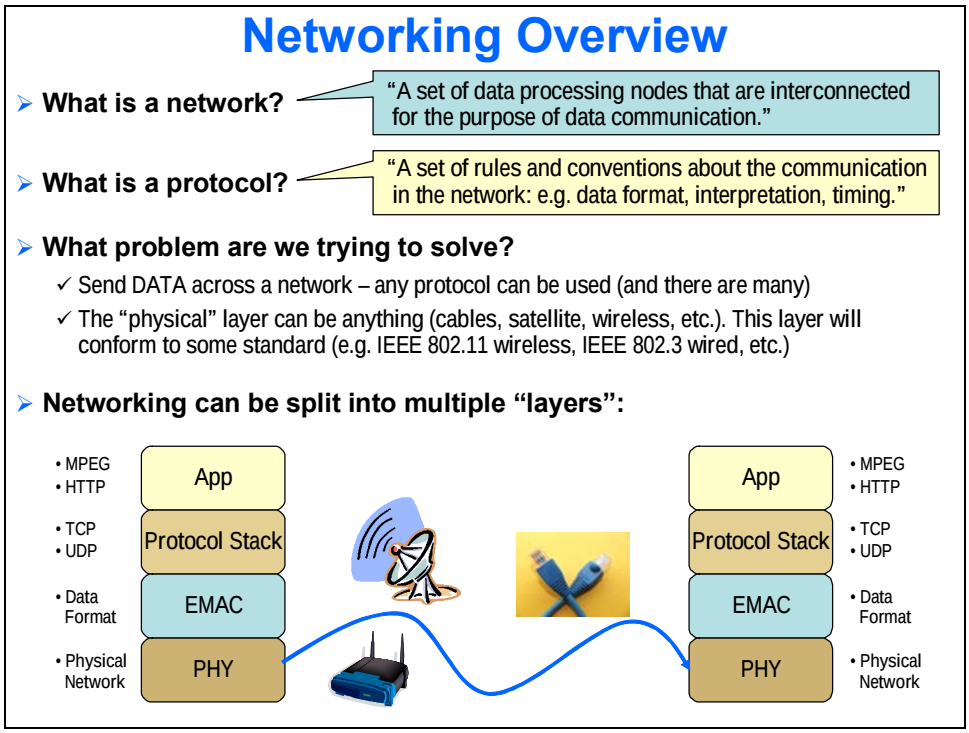
The Getting Started Guide does not cover all aspects and details about networking, but focuses mostly on how the Network Development kit developed for TI's Digital Signal Processors (DSPs) can assist embedded programmers to quickly develop networking software for a range of applications from servers (DAEMON or via sockets programming), clients, web development, ftp, etc.

We will begin with a general overview of terms of definitions and then proceed to giving an overview of the NDK. TI provides our own stack and example software that users can modify to fit their own system needs. One of the main goals of this app note is to take it from a user's perspective. You purchase the NDK from TI and you ask "now what?" This guide will attempt to enhance your "out of box experience" and avoid common problems/hiccups that first-time users experience.



2.2 Networking Overview

If you have extensive networking experience, the terms and definitions that follow will be a review. We are attempting to build a baseline of basic knowledge that we plan to build on throughout this rest of this guide.



Networking and communications are broad topics and cover a wide range of protocols, software and hardware. As stated above, a network is a set of data processing nodes that are interconnected in order to process data over small or large distances. On each end (source and destination), you will typically see an application running (client or server, e.g.). Most networking systems use a layered approach to interpreting and processing data that is transferred over a network. In this way, the application developer does not have to understand the entire software, protocols, stack, hardware peripheral (Ethernet MAC) and physical network. The application programmer is abstracted from these details which allows them to quickly write applications that suit their system needs, thus allowing the programmer to focus on their area of expertise – the application.

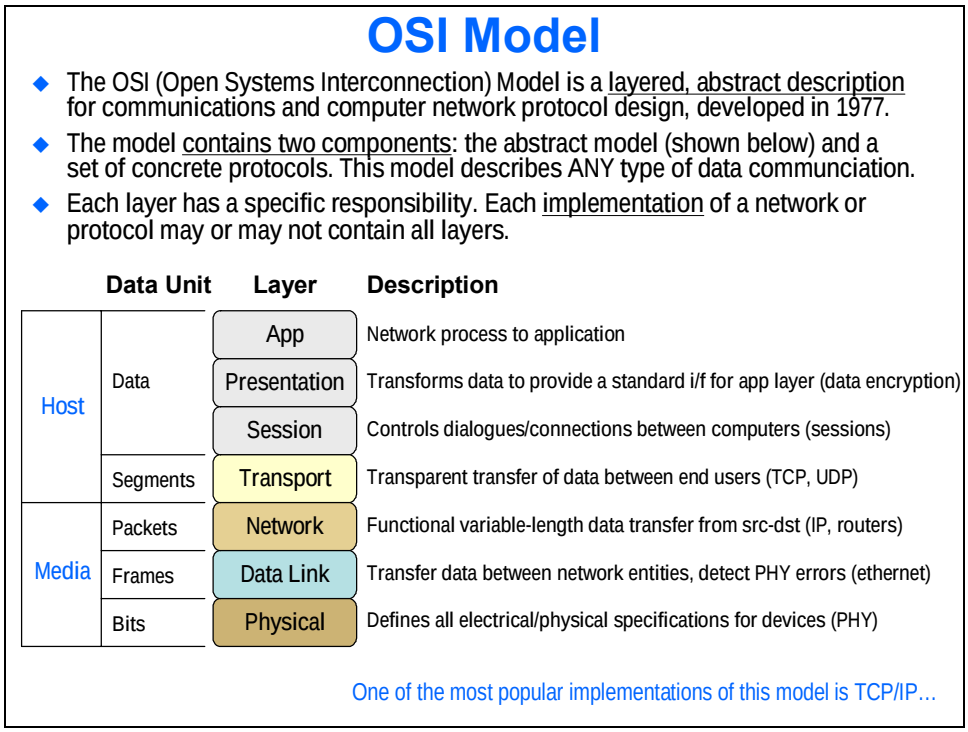
Most of the layers underneath the application are fairly rigidly defined. Depending on the processor of choice, the EMAC hardware may vary slightly and the actual physical connection (PHY) and how the data is transferred over the local area or wide area network can vary greatly – i.e. it could be over a wired connection (IEEE 802.3), wireless (IEEE 802.11), etc.

If every application programmer used their own protocol, EMAC hardware, etc., writing something that worked would be nearly impossible. So, standard protocols were developed in the late 1970s that defined the purpose of each layer to provide some standardization of how data communication should work. These protocols have been modified to keep up with today's technology to increase flexibility and enhance features.

2.3 OSI Model

In the late 1970s, this OSI model was developed to describe the function and purpose of each layer of a communication network. This is an abstract model – and every network implementation uses either the entire OSI model or a subset of it. As you can see below, each layer has a specific function from the physical layer that defines the actual electrical/physical specs for hardware devices all the way to the top layer – the application.

Any type of data communication can be implemented using this layered approach. The transport layer is a key area where protocols are defined and used. You may have heard of TCP or UDP – two of the more popular transport protocols. In this app note, we will focus more on these protocols as used by the data layer over Ethernet. We will describe in detail how TI's Network Development Kit (NDK) helps users develop application code that can use these protocols to communicate using these protocols over a network.



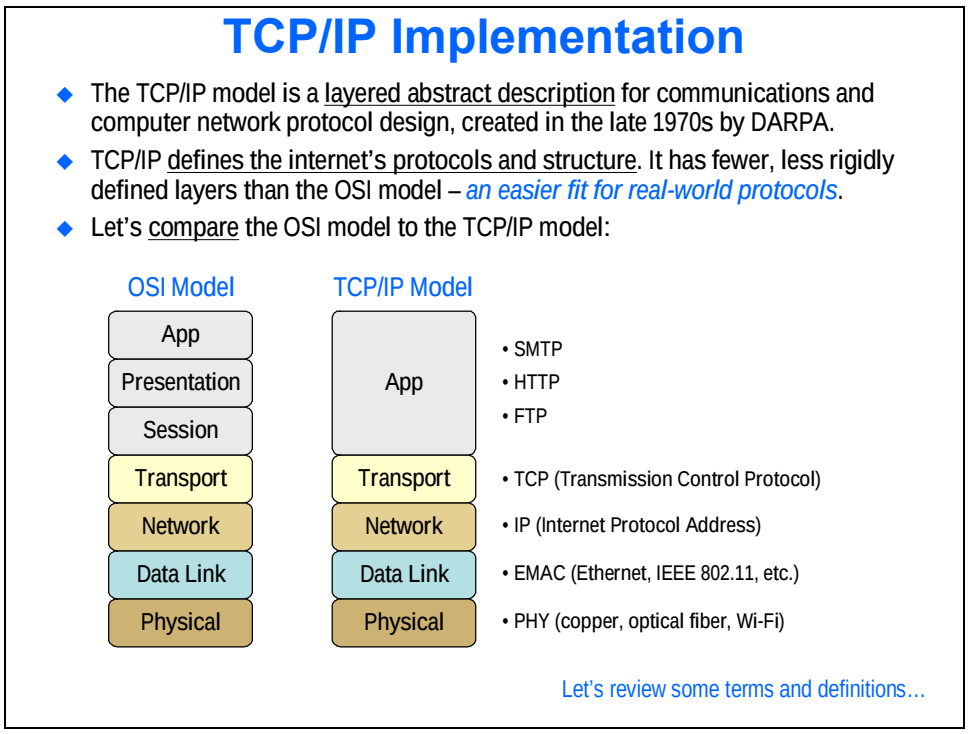
We will also investigate how the NDK incorporates several of these layers in order to allow the application programmer to focus on the application and not all the details in the lower layers. The NDK provides a set of libraries that use common sockets programming APIs to configure the lower layers rather easily.

Throughout this guide, we will continue to bring up this "layered" approach because the current implementations are all built upon this common model. It helps to know where and why these layers exist and how each layer is handled from the top to the bottom.

One of the more popular protocols we will investigate is TCP/IP (Transmission Control Protocol)/(Internet Protocol)…

2.4 TCP/IP Implentation of the OSI Model

This TCP/IP model was developed in the late 1970s by DARPA to describe and define the internet's protocols and structure. It has fewer, less rigidly defined layers which makes it easier to adapt to real-world protocols.



As you can see, TCP/IP combines the top three layers into one application layer – these applications might be SMTP, HTTP, FTP and others commonly used in the internet today like web browsers, audio and video players, chat, instant messaging, etc. The transport layer protocol has two variations: (1) TCP, which allows handshaking between the client and server and therefore guarantees the correct delivery of information; (2) UDP, which does not implement handshaking (therefore, it is less reliable) but has an overall reduced communications overhead (thus, a bit faster). Please note that both protocols do not guarantee delivery of data – i.e. if a wire is broken, no data is ever delivered – but only a node or computer using TCP will be aware of it.

IP is the logical internet address within the local network – i.e. a unique identifier for a computer or node inside a network. In wide area networks, the IP addresses can assume the same value among different local networks – i.e. the ISP might have a unique IP address, but every node underneath contains logical IP addresses to allow routing of the data to the correct node or computer on the network.

The data link layer uses an Ethernet MAC (Media Access Controller) that filters out any incoming frames not specifically destined to the node or computer on the network. To conform to the IEEE standard, all network equipment manufacturers are required to have unique sets of MAC addresses registered at the IEEE registration authority (http://standards.ieee.org/regauth). The physical network could be anything – copper, optical fiber or Wi-Fi.

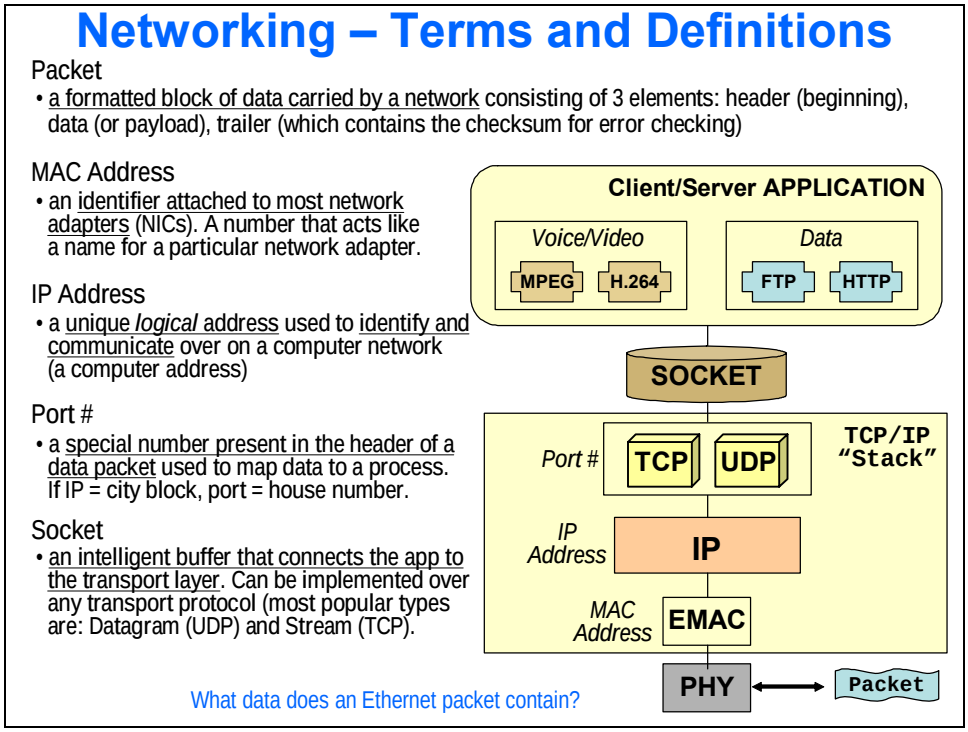
In this guide, we are focused mainly on Ethernet networking, the NDK and TCP/IP. So, let's look at some basic terms and definitions related to this more specific area of data communications…

2.5 Networking – Terms and Definitions

This is a view of the data flow between the layers. Let's start at the bottom right of the slide below. A packet arrives over the physical network. This packet contains three basic pieces of information: (1) the header (which contains the MAC address and other routing information); (2) the data itself (or payload); (3) the trailer – which contains a checksum for error-checking purposes.

The MAC address uniquely identifies a network adapter – e.g. the NIC card in your computer. The EMAC hardware processes the packet header and analyzes the MAC address. If the MAC address in the packet doesn't match the one configured on this specific adapter, the packet is not processed. In the upper layer (network), a second filtering takes place: the packet contains the IP address that is the logical identifier for the computer inside the network.

The packet also contains a port number. The port number is used to map the data to a specific application. Actually, these port addresses are unique for each application. HTTP uses one port number – FTP uses a different specific port number, etc.



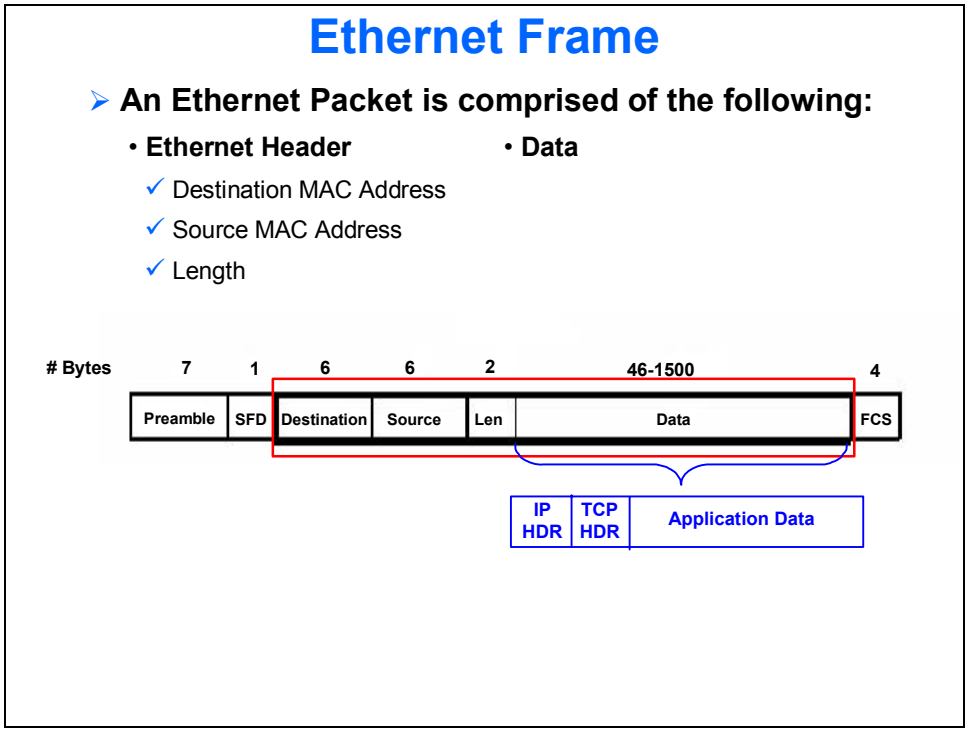
A socket is an intelligent buffer that is used to interface the application to the TCP/IP stack. The NDK provides common sockets programming APIs to allow the application programmer to interface to the TCP/IP stack in order to “open” a socket and transmit/receive data to the network.

We will investigate sockets programming later in this document.

Before we cover the NDK in more detail, we’d like to quickly cover some of the aspects of the EMAC hardware. To program this hardware manually requires a lot of time and expertise. If you plan NOT to use the NDK, you must become very familiar and intimate with each process inside the EMAC hardware and create your own stack (not easy, but it has been done by larger Communications Infrastructure companies). Using the NDK (kind of like a driver for the EMAC) is the quickest way to get your application running/receiving/transmitting network data.

2.6 Anatomy of an Ethernet Frame

An Ethernet frame contains 3 basic parts: (1) a header; (2) payload (data); (3) trailer (checksum). The key parts of the header are the source and destination MAC addresses and the length of the payload (data). All of this information is used to determine if this packet is designated for this specific EMAC device or not. If the packet’s IP address matches, the packet is processed and the data is read by the EMAC peripheral and transferred into memory (as specified by the peripheral’s configuration).



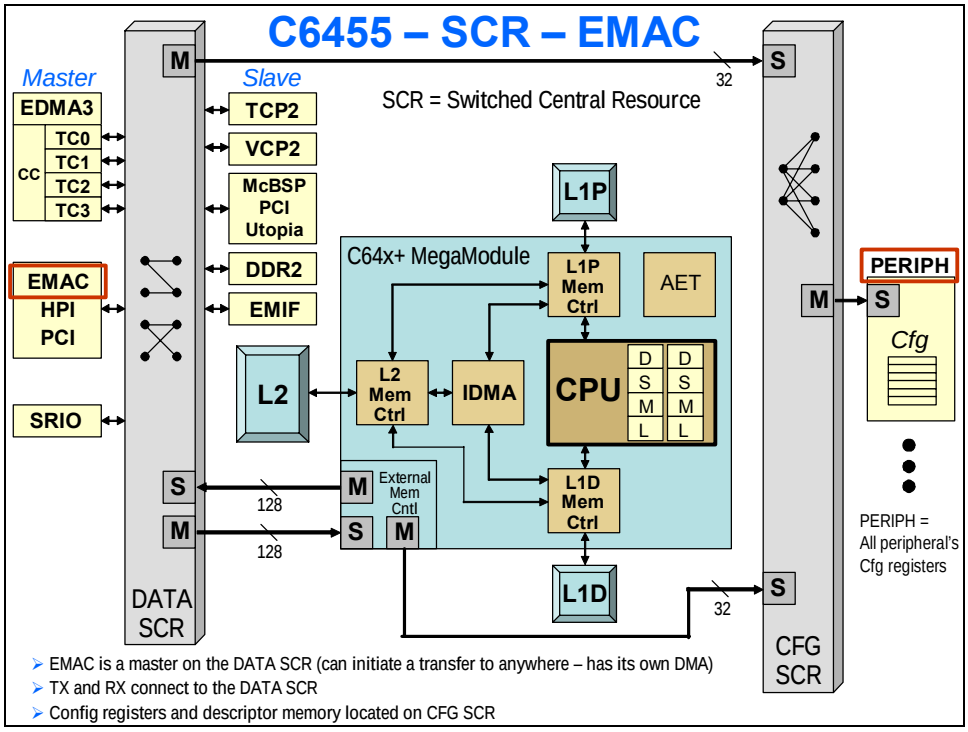
3 EMAC Architecture Overview

3.1 EMAC is a Master Peripheral

This is the architecture of the TMS320C6455 DSP. Each C6000 device that contains an EMAC may have slight differences from the picture below, but they are not relevant for a beginning user.

The Data Switched Central Resource (SCR) is a large bus switch that receives requests for resources that are tied to it. On the left are the master peripherals. On the right are the slave peripherals. The main difference between master and slave peripherals is that master peripherals can INITIATE a transfer – in other words, they have their own DMA controllers.

There are many details on this slide, but for the purposes of this discussion, the main point here is that the EMAC peripheral has its own DMA controller that can read/write to slave resources (e.g. DDR2 memory or internal L1/L2 memories).



3.2 Media Independent Interfaces (MII)

These terms – RMII, MII, GMII, and RGMII describe the size (# bits) and speed associated with the interface. Many architectural diagrams and datasheets just say “this chip has an RMII or GMII” – if you don’t know what they mean, then you could be confused. It is not necessary to understand each signal in the block diagram – this is outside the scope of this discussion.

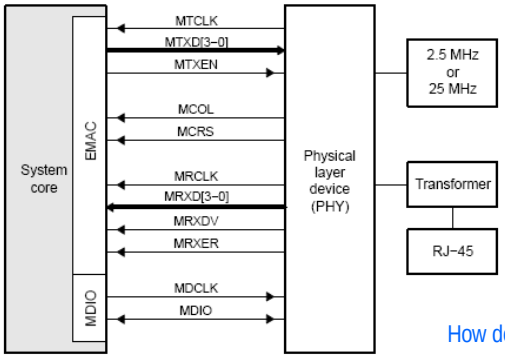
MII is the acronym for Media Independent Interfaces which specifies how many data bits and how fast it can run. For example, RMII is “Reduced” MII and uses 2-bit data and can run at 10 or 100 Mbps. The MII data signals are MRXD (for Rx) and MTXD (for Tx). The number of data bits depends on the type of MII interface. The MDIO signals are MDCLK and MDIO. These signals are used by the EMAC to interrogate and control the PHY.

Management Data Input/Output, or MDIO, is a bus structure defined for the Ethernet protocol. MDIO is defined to connect Media Access Control (MAC) devices with PHY devices, providing a standardized access method to internal registers of PHY devices.

These internal registers provide configuration information to the PHY. This bus allows a user to change configuration information during operation, as well as read the PHY's status. It is a standard-driven, dedicated-bus approach that's specified by IEEE workgroup 802.3. The MDIO interface is implemented by two pins, an MDIO pin and a Management Data Clock (MDC) pin. This standard is available for all speeds of Ethernet.

Media Independent Interfaces (MII)

- **C6455 EMAC supports 4 types of interfaces:**
  - RMII – Reduced MII (2-bit data, 10/100 Mbps)
  - MII (4-bit data, 10/100 Mbps), shown below
  - RGMII – Reduced GMII (4-bit data, 10/100/1000 Mbps)
  - GMII – Gigabit MII (8-bit data, 10/100/1000 Mbps)
- **EMAC: data interface between PHY and system core (clk, control, data)**
- **MDIO: used to interrogate/control Ethernet PHY**



How does the EMAC work?

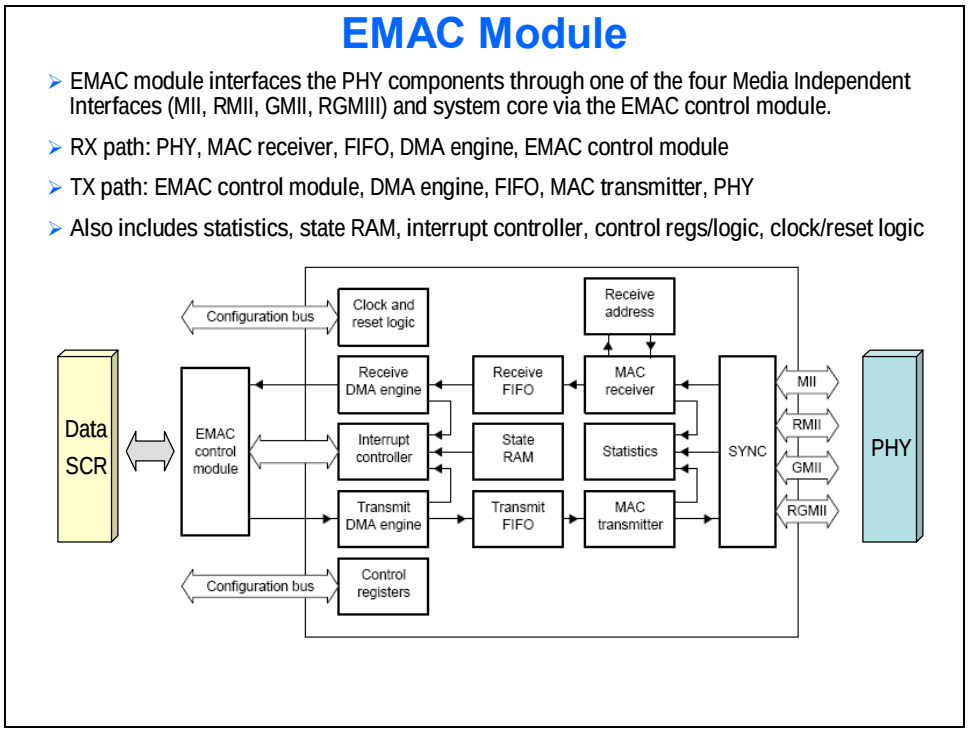
3.3 EMAC Module

This is a quick overview of how the EMAC peripheral works. There are several blocks that will require further investigation by the user – however, the point here is to have a basic understanding of the receive and transmit paths and the presence of two DMA engines – one for receive and one for transmit.

For receive, the packet arrives at the PHY device. The MAC receiver parses the data and transfers it to the receive FIFO. The receive DMA engine then makes a request to the EMAC Control Module (and then to the Data SCR) to transfer this data from the FIFO to a resource tied to the Data SCR. No CPU involvement is required because the EMAC contains its own DMA controller that can initiate transfers of data independent of the CPU.

The transmit side is similar – just back the other way. The EMAC control module requests a transfer from a Data SCR resource to the transmit FIFO. The transmitter then compiles a packet with the data and sends it out via the PHY device.

Other blocks are busy gathering statistics, managing the clocks and configuration registers as well as optional interrupts that can be sent to the CPU. If a user didn't have the NDK, they would need intimate knowledge of each configuration register, buffer descriptors, etc. in order to program the EMAC to perform a transfer. While it is possible to do this manually, having the NDK services makes life much easier.



3.4 EMAC Data Layer – Example (TX)

Here is a brief look into how a packet might be transmitted to the PHY. The EMAC uses buffer descriptors to contain the proper information needed for a transfer – kind of like a DMA configuration specifies source, destination and length information. There is a total of 8K bytes of descriptor memory for both Rx and TX (512 descriptors at 16 bytes each). A single buffer descriptor is 16 bytes long and contains the following:

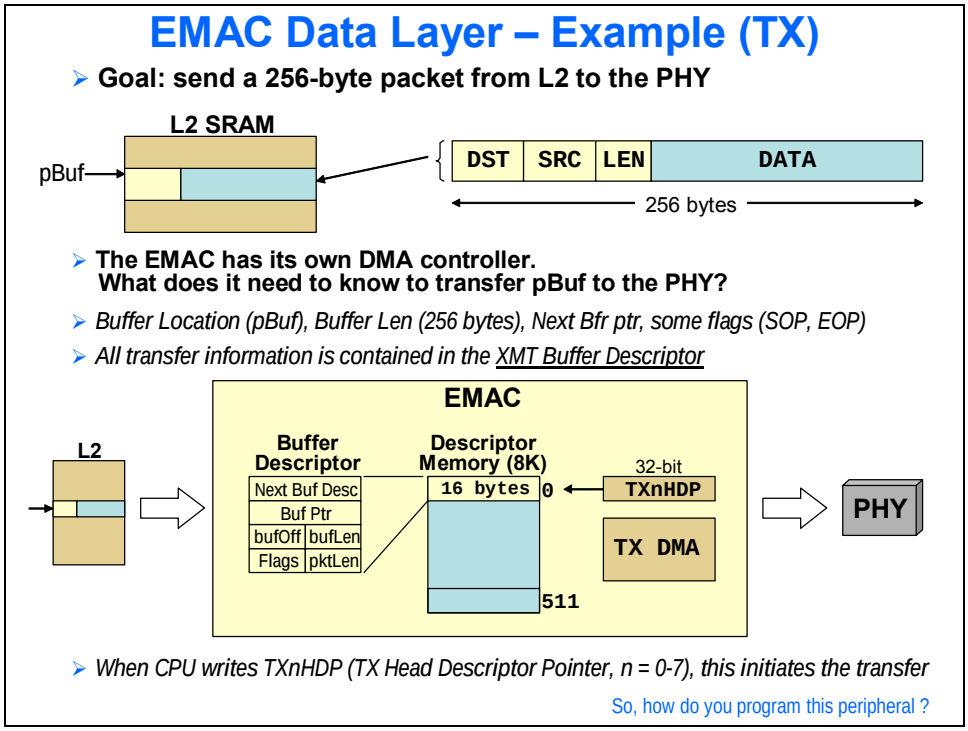
Next Buf points to the next descriptor if you have one. *Buf Ptr* contains pBuf (the source of the transfer). *bufOff* is the offset from Buf Ptr to the first piece of valid data (if necessary). *bufLen* is the length of the buffer (256 bytes). *pktLen* is the length of the entire packet (multiple descriptors).

Flags contain information/status about the packet and transfer.

The CPU has to fill in the descriptors before the HDP (head descriptor pointer) is written to. The HDP is the "Head" (as opposed to the tail) of a linked list of descriptors. When the "head" is written to, this initiates the transfer (via the EMAC's DMA) from the L2 memory to the transmit FIFO (and eventually to the PHY).

When the TXnHDP is written to, this initiates the transfer from L2 (in this case) to the EMAC FIFOs. The EMAC then processes it (adding CRC, etc) and then sends it to the PHY. This EMAC processing is automatic – no user intervention is required.

Again, the point here is that the NDK can perform all of this configuration and processing for you vs. doing it manually. For most users, this is a huge time savings in their application development.



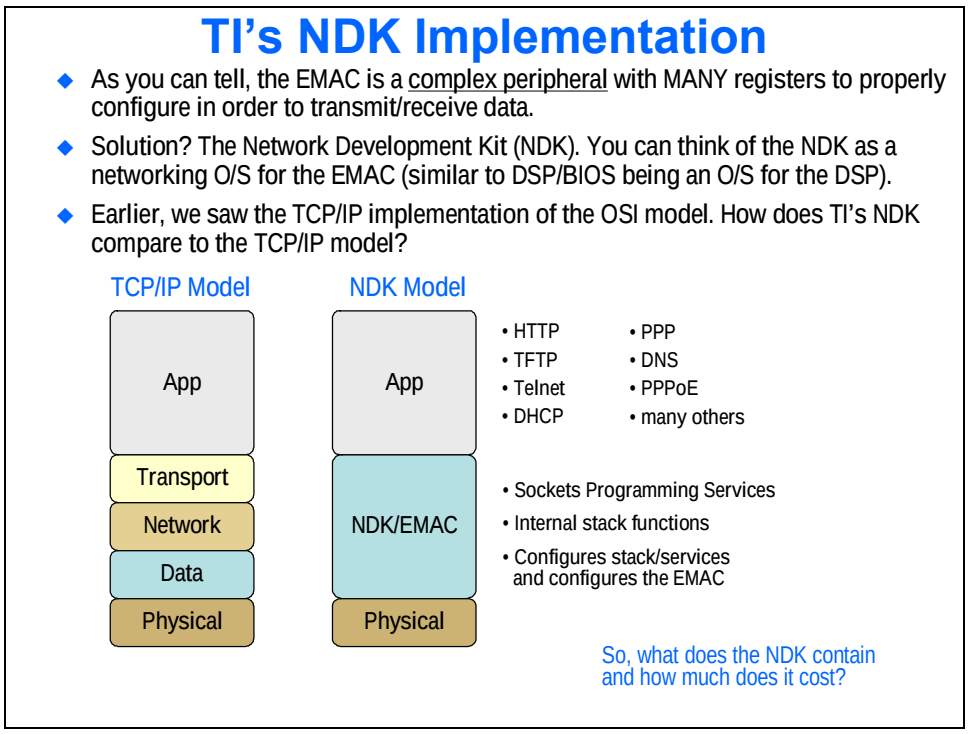
4 Network Development Kit (NDK)

4.1 TI's NDK Implementation

The NDK uses a layered approach similar to the TCP/IP model. As is shown, the App and physical layers are still present and the NDK model contains the transport, network and data layers.

The goal here is to make life simple for the programmer. Would you rather spend time looking up 40-50 configuration registers and the proper bit settings to get this right or look up a few supported APIs and use them or, simply modify an existing working example to perform exactly what your application needs?

The heart of the NDK is the TCP/IP stack. It also contains configuration of services such as HTTP, telnet, DHCP, etc. along with supporting the standard sockets programming APIs. If you've done sockets programming before, the NDK world will look very familiar to you.



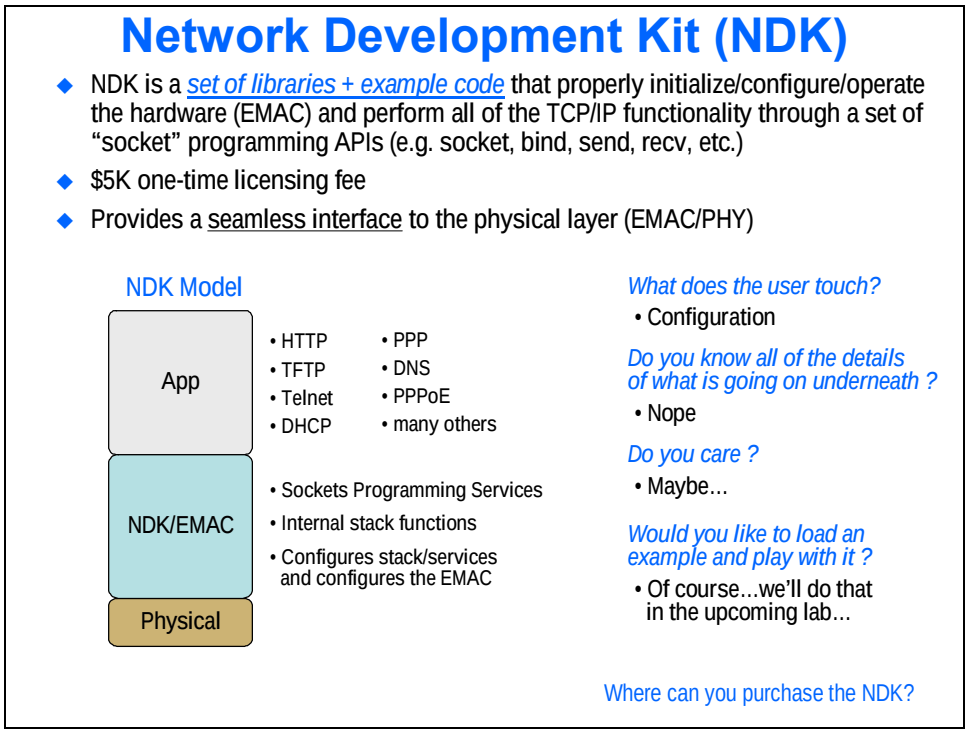
4.2 Network Development Kit (NDK) Contents

The NDK is a set of libraries and example code that properly initializes and configures the EMAC peripheral to perform whatever service is required by your application.

When installing the NDK, it is important to note that two separate downloads are required: (1) the stack (platform-independent set of libraries, code); (2) the examples (platform-dependent NDK support package). Many customers just download the stack and wonder “where did client.pjt go?”. Well, client.pjt and many other examples are contained in the 2nd download that is specific to the board/processor you are using.

The platform-dependent NDK support package is the software set that contains the example projects and all the hardware libraries for each development kit supported by NDK. The user should refer to www.ti.com for latest support for each device/development board.

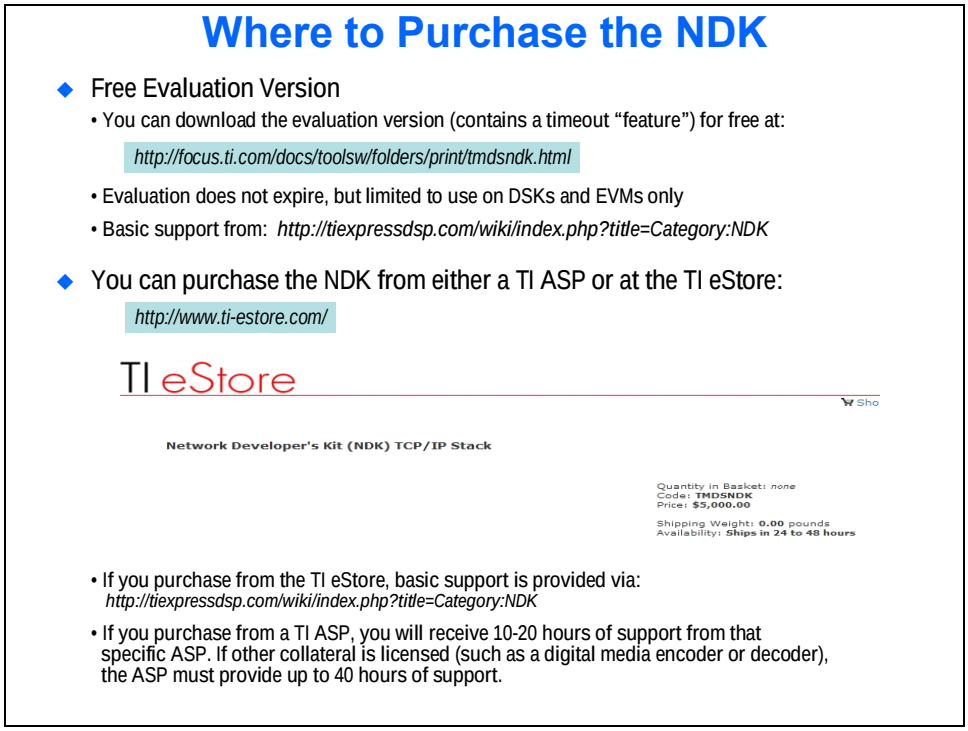
In other words, when you install NDK, you get all the core stack libraries, the documentation, the main files for the examples (client.c, cfgdemo.c, helloWorld.c), the server files located at <examples\tools\common\servers>, all the other nettools (webpage.c, console, hdlc, cgi) and the winapps folder (used to test communications). However, the main board-specific client example is NOT installed. The examples only get installed with the 2nd download – the platform-dependent NDK support package.



4.3 Purchasing the NDK

You have several options when it comes to downloading the NDK. First, you can use the evaluation version provided at the URL listed in the slide below. It is a full working version of the NDK. However, it contains a timeout “feature” – if the stack runs for 24 hours, it resets itself.

If you plan to purchase the NDK, you can do so by visiting the TI eStore or one of TI’s Authorized Software Providers (ASP). If you purchase from the eStore, basic email/phone support is provided. However, if you purchase from a TI ASP, you will receive 10-20 hours of support from that specific ASP. If you purchase other collateral from that ASP, your support will increase even further.

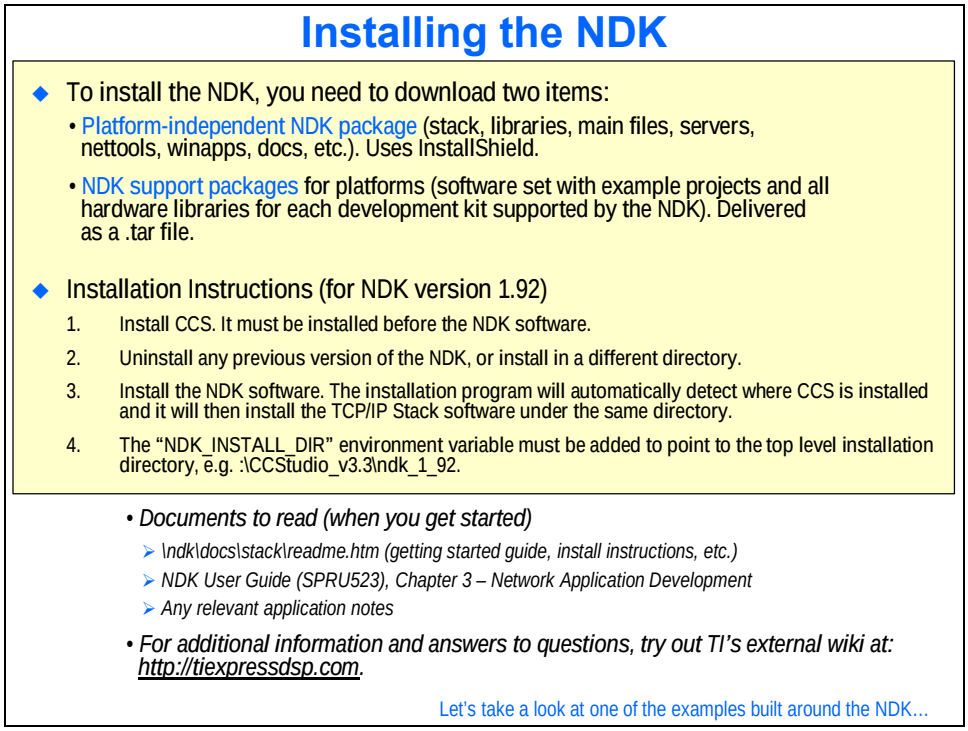


4.4 Installing the NDK (some highlights)

Here is a brief summary of the procedure for installing the NDK. Just following these simple instructions can help you avoid headaches in the future. One of the more common mistakes users make is forgetting to set the NDK install environment variable – very important – in fact, if you don't, nothing works.

Also, do something that is very counter intuitive – READ the readme file. How many times do we skip reading the readme files? Well, this is one you don't want to skip.

Also, as documented in one of the upcoming labs, as soon as you install the platform-dependent examples, make a copy of the /examples directory. This may come in handy if you modify something and can't get it back to the original state.



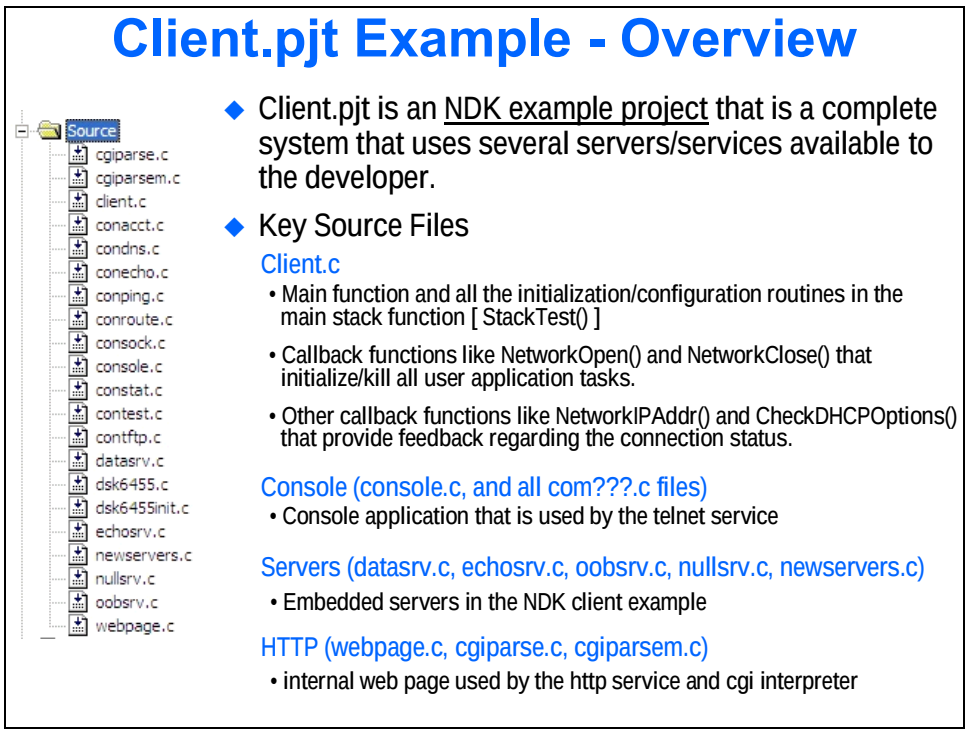
5 Client.pjt – Your "StarterWare"

5.1 Client.pjt Example – Overview

Once you have download the platform-specific Network Support Package, you'll find in the directory structure a project named client.pjt. There are a few others, but we'll focus on client.pjt in this guide.

In Code Composer Studio (CCS), when you open this project, you'll see a list of source files. Client.c is the main source file that contains the stack initialization and configuration routines. The other source files are described in brief in the slide below.

Client.pjt may be a misleading name because it contains services like HTTP, telnet, clients AND servers. So, a better name might have been "ndk.pjt" or "everythingButTheKitchenSink.pjt". But, client.pjt is what it is, so let's get past the name and move on to see what we can do with this example code…



5.2 Client.pjt – What Does it Contain?

It is best to start with a working example and then modify it to do exactly what your application needs. For example, you could start with client.pjt and then remove services down to a bare minimum – e.g. just to perform a DAEMON echo server. Or, you might want to do a sockets programming echo server.

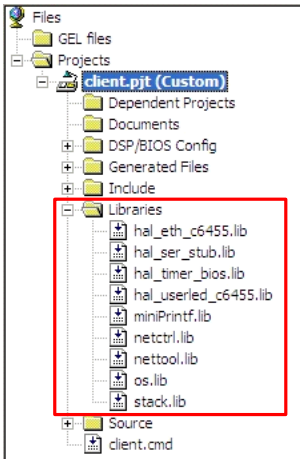
In one of the first labs you will run, you will open client.pjt, build it and then run it and observe all of the available services. In a later lab, we'll document step-by-step how to modify client.pjt to only do a DAEMON server. For now, let's see what is inside client.pjt and ask the question – which files/libraries are essential and which can be removed?

First, all NDK examples use TI's real-time kernel DSP/BIOS. All services are TSKs in the system and contain priorities and therefore can be scheduled by DSP/BIOS. If you open up the .tcf file, you'll be able to view the DSP/BIOS settings along with cache and other memory settings.

The next major piece of the NDK are the libraries (explained in more detail on the next slide). All libraries are essential to the NDK operating correctly, so please don't remove any library files. Some source files can be modified and we'll deal with those shortly.

Client.pjt – What Does It Contain?

- When you purchase the NDK, you get several examples. The key example that you can modify is client.pjt. So, it is thrown in your lap…now what?
- The question is: what do we need to keep and what can we delete? Remember, our goal is to receive a packet and echo it back.
- Let's first look at what the client project contains:



- Client.pjt contains
  - DSP/BIOS .tcf file
  - Include Files
  - Libraries
  - Source Files
  - Command File (.cmd)
- Note: client.pjt contains client AND server software

What do the different libraries do and which ones can we delete?

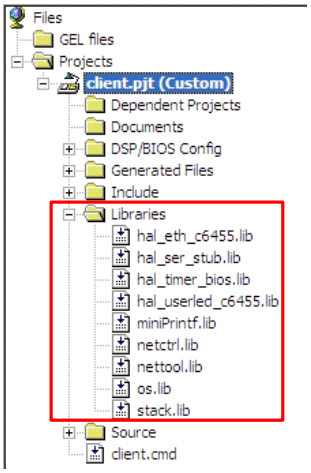
5.3 Client.pjt – Libraries

Shown below is a short description of each library and what they contain. If you want to dive in deeper, there are more detailed explanations in the NDK User Guide (SPRU523).

Again, which libraries can we remove? None. They are all important. Update – revision 1.94 of the NDK allows users access to the source code for the nettool library, so you can remove unwanted services and recompile that library to save some code space.

Client.pjt - Libraries

- The NDK Libraries are where all of the work gets done. When you call an API in source code, it will activate one or several of these libraries to perform the requested operation and properly configure/run the EMAC hardware.



- HAL??.lib (Hardware Abstraction Layer)
  - Interfaces the hardware peripherals to the NDK
  - Including timers, LED indicators, ethernet devices, serial ports.
- MINIPRINTF.lib
  - Provides small-footprint printing functions
- NETCTRL.lib
  - This is the MANAGER of the stack. It controls interaction between the TCP/IP stack and the outside world
- NETTOOL.lib
  - Contains all sockets-based network services + a few add'l tools to aid in developing network applications
- OS.lib
  - Adaptation layer that maps O/S function calls to DSP/BIOS function calls. Includes thread mgmt, mem allocation, packet buffer mgmt, printing, logging, sections, cache coherency
- STACK.lib
  - Main TCP/IP networking stack. Contains everything from the socket layer at top to the Ethernet/PPP layers at bottom.

So, which libraries can we delete? None. They are all important. Now, let's do some labs…

6 Lab 1: Code Composer Studio (CCS) Setup

Lab 1 – DSK Hardware/Software Setup

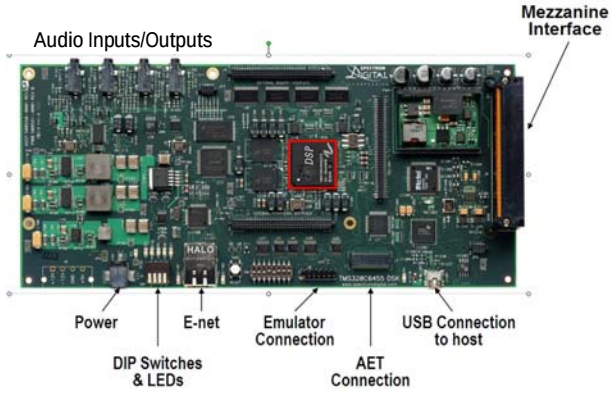
Software

1. Run CCS Setup
2. Start CCS
3. Configure CCS Options
4. Close CCS

Hardware

1. Verify hardware setup
2. Supply power & verify connection

Time: 15 minutes



6.1 Lab Overview:

- All labs and code are based on the C6455 DSK. When you download the starter files, they will only build/run on the C6455 DSK. The Mezzanine card (for use with Serial Rapid I/O) is not required.
- The goal of this lab is basically just to make sure Code Composer Studio (CCS) and your hardware are set up properly. If you have minimal experience with CCS, then take the time to walk through this lab. In lab2, you'll get to examine the audio pass-through lab and how it works.
- For all labs, we have used the following setup: CCS 3.3, DSP/BIOS 5.31, NDK version 1.92, C6455 DSK. You can use newer versions of these tools – just be aware that the author has not tested out these labs on the newer versions.
- If your setup is different than above, you may need to modify some of the steps in the labs to match your specific setup.

6.2 Hardware Setup

Please note, the instructor may have already done these steps for you. Raise your hand and ask which steps have already been completed. These first few steps are here just for completeness in case you get home and need some assistance setting up your own board.

1. **Connect the USB cable between the C6455 DSK and the PC.**
2. **Connect the Audio cable to the DSK's "LINE IN" jack.**

   We will connect the other end to the PC after we complete the sound test on the PC.
3. **Connect the headphones to the DSK's "LINE OUT" jack.**

   If you have a Y-adapter, plug this into the "LINE OUT" jack on the DSK. You may also use "HEADPHONE" jack on the DSK, but it has much lower volume.
4. **Connect power to the C6455 DSK.**

   When power is applied, D3 & D4 LEDs should be lit.

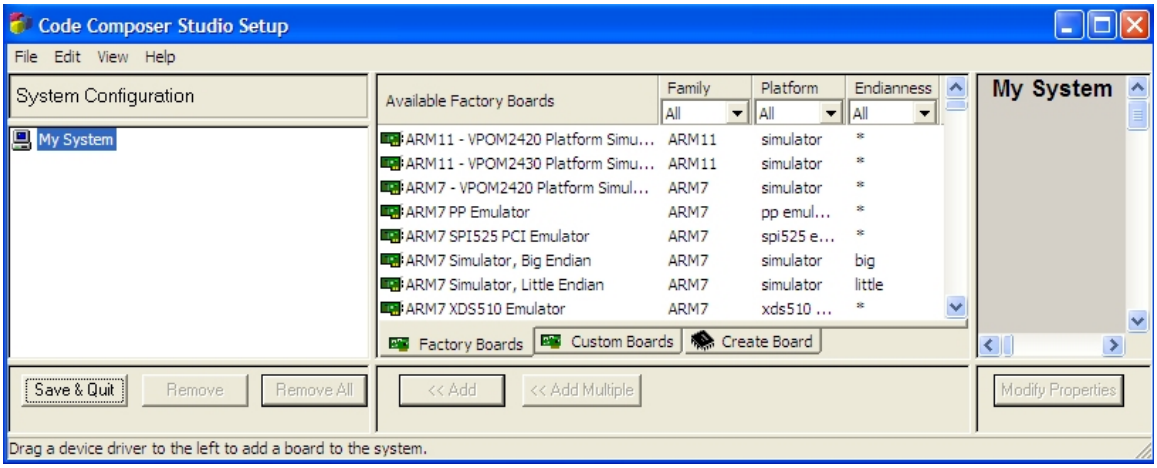
6.3 CCS Setup

The lab workstations should have been pre-configured with all the necessary software tools. While Code Composer Studio (CCS) has been installed, you will need to assure it is set up properly. CCS can be used with various TI processors – such as the C6000 and C5000 families – and each of these has various target-boards (simulators, EVMs, DSKs, and XDS emulators). Code Composer Studio must be properly configured using the CCS_Setup application.

5. **Start the CCS Setup utility using its desktop icon:**

Be aware there are two CCS icons, one for setup, and the other to start the CCS application. You want the **Setup CCStudio v3.3** icon.

6. **When you open CC_Setup you should see a screen similar to this:**



Your screen might look different if there were any previous boards installed from previous classes or work on your student machine.

7. **Clear any old system configurations.**

   If there are any boards/simulators listed under *My System* under **System Configuration**, click the Remove All button to clear the configuration.

8. **Select the proper DSK6455.**

   Use the filter in setup. Choose Family = C6455. There should be only 2 boards showing: C6455 DSK and C6455 DSK with Mezzanine. Click the board that says C6455 DSK and then click Add. Then select Save & Quit and answer “yes” to startup CCS on exit.

Note: if the DSK “cannot connect to target”, close CCS, hit the white reset button on the DSK (w/pwr on) and re-invoke CCS.

6.4 Set up CCS – Customize Options

Section Overview: There are a few option settings that need to be verified before we begin. Otherwise, the lab procedure may be difficult to follow. The following bullets are HIGHLIGHTS for what you will do in the numbered steps.

- Disable open Disassembly Window upon load
- Go to main() after load
- Program load after build
- Clear breakpoints when loading a new program
- Connect automatically to the target

9. Connect to the target (using Alt-C or Debug → Connect).

When you connect to the target, watch the symbol in the lower LH corner of CCS change colors and reflect that you are connected to the target.

10. Use the Customize Dialog box to set specific options.

Select:

Option → Customize…

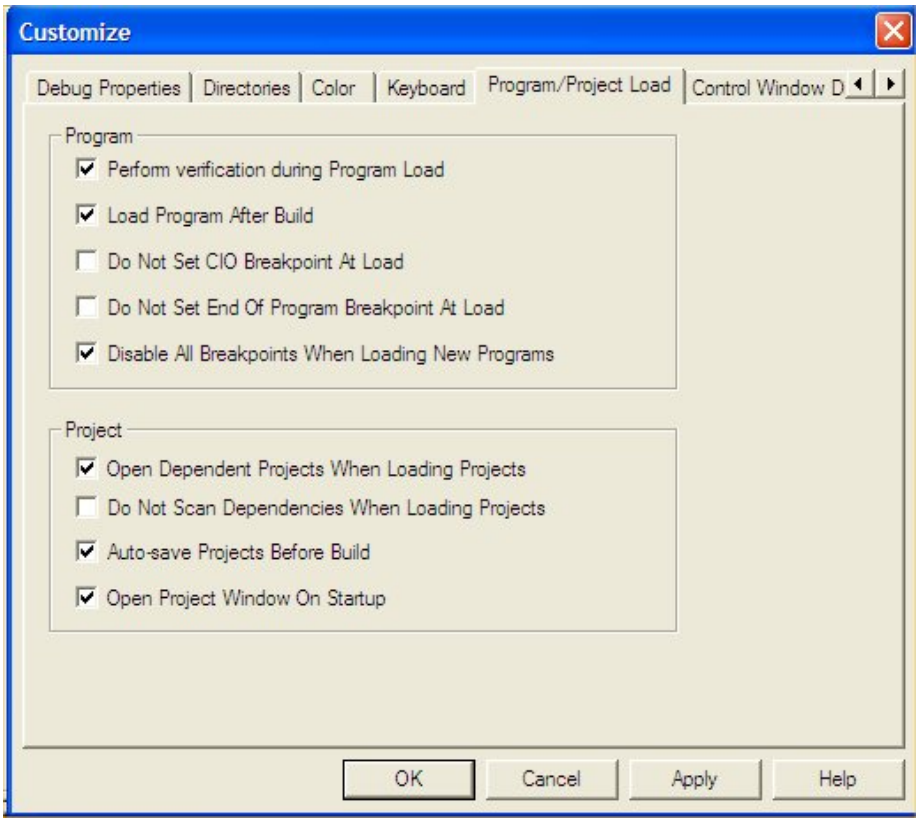
Under the tab *Debug Properties*, uncheck the box for *Open the Disassembly Window* automatically. Check the *Perform Go Main automatically* box. Check the following check box: *Connect to the target at startup when a control window is open*.

If there is anything else checked, just leave the settings as they are.

11. Set Program Load Options

On the "*Program/Project Load*" tab, make sure the options shown below are checked:

- Load Program After Build
- Auto-save Projects Before Build
- Disable All Breakpoints When Loading New Programs



By default, these options are not enabled, though a previous user of your computer may have already enabled them. If any other boxes are checked, leave these settings alone.

Click OK.

Conceptually, the CCS Integrated Development Environment (IDE) is made up of two parts:

- ***Edit*** (and Build) programs (uses editor and code gen tools to create code).
- ***Debug*** (and Load) programs (communicates with DSP/simulator to download/run code).

The *Load Program After Build* option automatically loads the program (.out file) created when you build a project. If you disabled this automatic feature, you would have to manually load the program via the File→Load Program menu.

You might even think of an IDE as standing for Integrated Debugger Editor, since those are the two basic modes of the tool

---

12. Make sure CCS is connected to the board.

Look in the lower LH corner of your CCS window. Hover your cursor over the symbol in that corner. If the symbol is green and the box says “connected to target”, proceed to the next step to test the connection. If the symbol is not green and doesn’t say “connected to target”, do the following:

Debug → Connect (or “Alt-C”)

When you perform the connect, you should see the symbol turn green and a box that says your target is now connected.

13. Load in the test.out file

On the CCS menu bar, select:

File → Load Program… (or “Ctrl-L”)

navigate to: `C:\IW64x+\6455_DSK_Testfile\`

Load: audio_app_lab6.out

If CCS says it can’t find “main.c”, just ignore this and click NO (you don’t want to browse for it).

14. Run the audio test file.

You can run by left-clicking on the “Run” button in CCS (located on the LH side):

Or, you can simply press F5 (or choose Debug → Run).

15. Launch a music file

Double click on any *.mid file under `C:\IW64x+\music`

Confirm that music is playing on the PC (either via the PC’s speaker or plug your headphone into the PC’s headphone jack). Make sure the media player is set to “repeat” or “play forever”. This is important because sometimes you think you’re getting “no output” when it was a simple issue of your music (input) stopped playing.

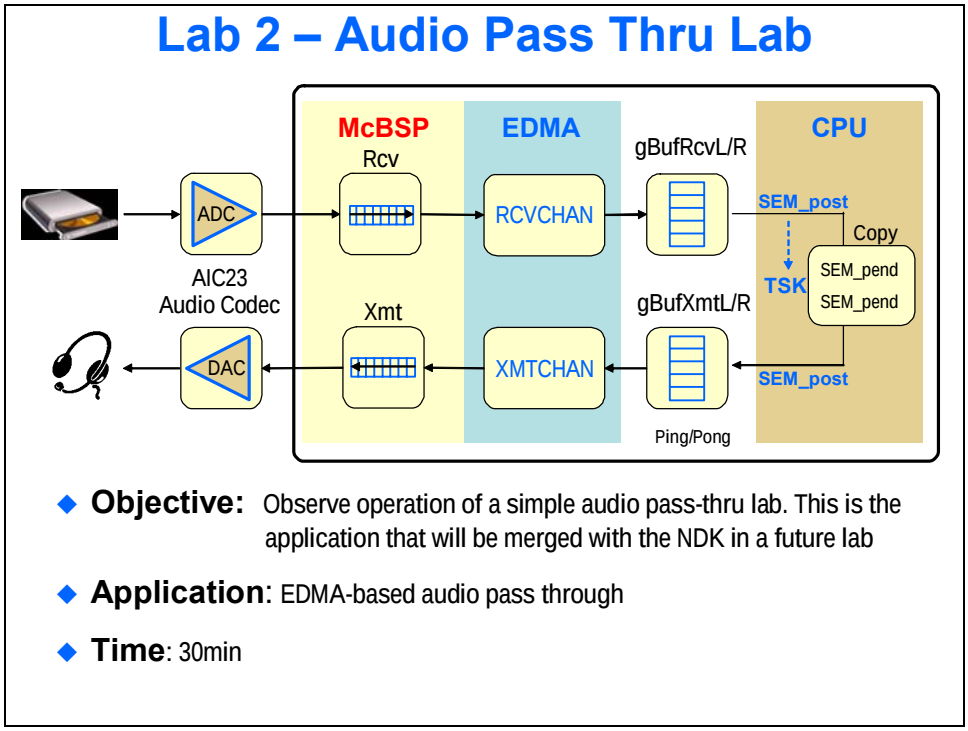
Connect the DSK’s LINE IN to the PC’s headphone out. Make sure your earbuds/speakers are connected to LINE OUT on the DSK.

Confirm that music is playing via the DSK’s headphone out jack (either by using your headphones or earbuds). If not connected yet, plug in the Y-adapter and a set or two of the available earbuds. You should also see LED D10 blinking while the audio is running.

If the music is NOT playing, contact your instructor for assistance. If music IS playing, you have successfully set up CCS and connected the DSK correctly.

You’re Done

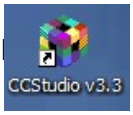
7 Lab 2: Analyze Audio Pass Thru Application



7.1 Lab Overview:

- In a future lab, we'll be adding an existing audio application to the NDK. This is your chance to familiarize yourself with the basic operation of this application.
- If you have minimal experience with DSP/BIOS, Code Composer Studio and the EDMA, this is a great lab to walk through to help you understand these basic concepts.
- In the 4-day IW6455 workshop, you actually build this example code from the ground up.
- If you don't have a C6455 DSK, you may need to look at using a different example that comes with your development board – preferably one that uses DSP/BIOS. The example code that is required for this lab only works on the C6455 DSK.

7.2 Open the Project, Build/Run

1. **Power cycle the DSK and start CCS using the desk** 

.
2. **Reset the C6455 DSK**

   Select:

   GEL → DSK6455 Functions → Reset

3. **Open the project and observe the contents.**

   Select:

   Project → Open

   and navigate to:

   `C:\IW64x+\solutions_ti\lab6b\`

Open the file: audio_app.pjt

FYI – this is the solution to lab6B which you end up building from scratch in the 4-day IW6455 workshop (hey, sign up today !). Anyway, this will be a quick overview of the application – just skimming the surface, so buckle up.

Take a moment to look at several critical source files and observe their contents/purpose:

- **main.c** – buffers declared, EDMA setup, enable interrupts, init buffers to zero
- **edma.c** – setup for EDMA channels to copy music data from McBSP to/from buffers
- **mcbsp_aic23.c** – set up for McBSP using register layer chip support library (CSL)
- **tasks.c** – tasks to manage buffers and processing (oh, and toggling the DSK's LED)

WARNING:

Please note that this lab (and any future labs) will NOT work until you have installed the latest Chip Support Library (CSL) for the 6455 DSK. The CSL version that was used to build these files was:

CSL_6455_v3.00.10.02

You will need to download that version or a later version from the TI web site. It was too large (about 16M zipped) to ship with the starter lab files.

Build the project.

Select:

Project → Rebuild All

Or…

Click on the Rebuild All icon:

CCS will compile all files from scratch.

Note the progress of the build in the **Output window** at the bottom of CCS. When done, the output file should automatically be loaded to the DSK target, as indicated by the **Loading Program** popup window that should briefly appear during the download. If errors are reported, recheck the steps above and try again. If there are still errors and you don't know how to fix them, ask the instructor for assistance.

7.3 Debugging Techniques

4. **Run to Main (if not there already).**

   In Lab 0, we also enabled the option to automatically go to main when a program is loaded. So your program should be sitting at main( ) right now. If you are not at main(), and the program has been loaded, run to the main function using:

   Debug → Go Main (or Ctrl-M)

   The debugger should run past the system initialization code, until **main( )** is reached. Since main( ) is in main.c, this file should appear in CCS's main work area. Many initialization steps occur between reset and your main program.

5. **Start music playing.**

   On the PC, **begin playing music** as the input to the DSK via the audio patch cable. Turn on the output speaker, or prepare to listen to the headphones. The music files can be found in the **C:\IW64x+\music** folder (actually, any MP3 or .midi file will work).

6. **Run the program.**

   Start the program running via **Debug → Run**, function key **F5**, or the  icon:

7. **Verify that the selected music is now playing from the output device.**

   If not, contact your instructor.

8. **Halt the program execution.**

   To halt the running program, use:

   Debug → Halt

   or the function key **Shift+F5**, or the halt icon:  . After testing the halt function, resume the program by asserting **run** again.

7.4 Analyze TCF (Text Configuration File)

The .tcf file is used to set up various system constructs including memory, DSP/BIOS Scheduling, semaphores, PRD functions, etc. Let's look at each one of these briefly. In the next lab, you'll have the opportunity to create your own .tcf file from scratch. For now, we just want to observe the contents of an already written .tcf file to become familiar with it.

9. View System – Global Settings.

Click on the + next to *DSP/BIOS Config* and open **audio_app.tcf**. As you can see, there are multiple items you can set up using the .tcf file. Let's look at them in order.

Click on the + next to *System*. Right-click on *Global Settings* and select Properties. As you can see, this is where you configure the target, DSP frequency, endian mode, and some Real-Time Analysis choices. Click OK.

10. View MEM – Memory Section Manager.

Click on the + next to *MEM*. Notice the memory areas that are configured. Click on *IRAM* and observe the base address and length. This is actually the default memory map for the DSK.

Let's view the properties of IRAM – right-click on *IRAM* and select properties. Do not change any values – but this is where you can change the properties of each memory area. Click cancel.

11. View Instrumentation

Click on the + next to *Instrumentation*. The LOG manager allows you to create a LOG object and perform a LOG_printf() to the CCS output screen. This is great for debugging vs. using a standard printf(). STS is used for statistics. You can benchmark any piece of code you like using a statistics object and a few commands – again, we'll do this in a later lab.

12. View Scheduling

- Click on the + next to *Scheduling*. This is where all of the DSP/BIOS Scheduling constructs are configured – for example, SWIs, TSKs and PRD functions.
- Click the + next to *PRD*. Notice we have one PRD function in the system. This PRD function runs every ½ second to toggle LED0 on the DSK. Right-click on PRD_waitPRD and select properties. Observe that the period (in ticks – i.e. 1ms) is set to 500 = ½ second. Click Cancel.
- Click on the + next to *HWI*. This is where you can map interrupt sources to the 12 CPU interrupts. Click on the "-" next to HWI.
- Click on the + next to TSK. Notice that we have 2 TSKs running in the system (other than the default TSK_idle): TSK_DipLED and TSK_processBuffer. Right click on TSK_DipLED and select properties. Select the Function tab. Notice that the function associated with this TSK is _tskDipLED. Click cancel.

13. View Synchronization.

Where is the waitPRD semaphore configured? Click on the + next to Synchronization. Click on the + next to SEM. Notice there are 3 semaphores configured in this system. One of them is waitPRD. You will gain more experience with these in later labs.

14. Close audio_app.tcf and do not save changes.

15. Build, load and run the program.

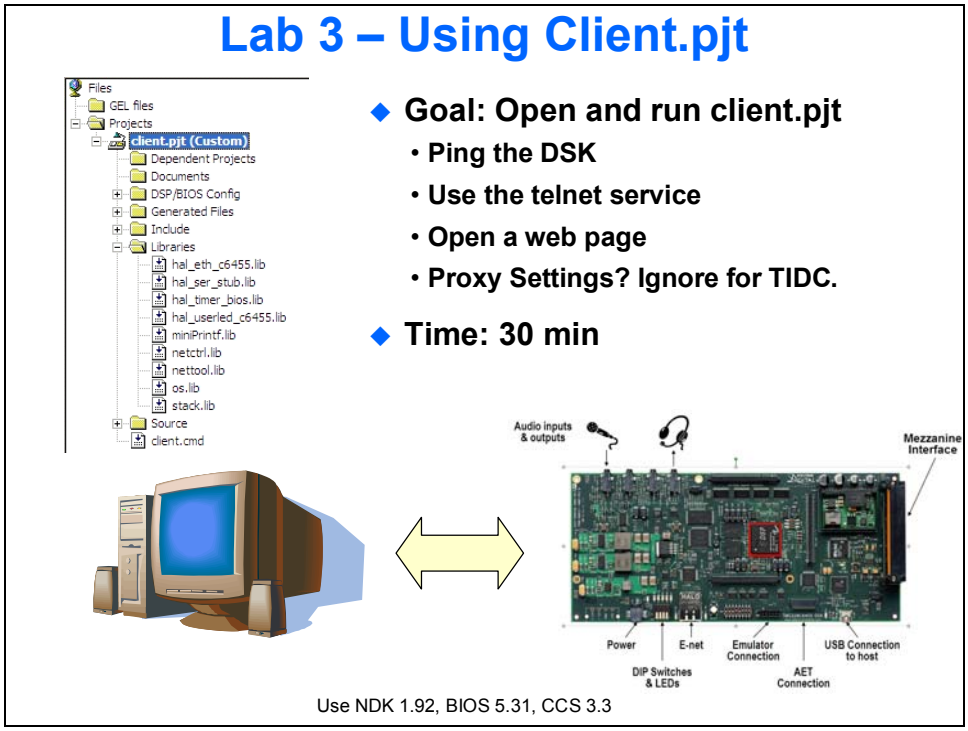
First, get some music playing (can be found in C:\IW64x+\music folder).

Next, click the Rebuild All button toward the top middle of your screen (three red down arrows). After the program loads, click Run (left hand side – looks like a runner). If audio is playing into the DSK, you should hear the audio in the headphones.

The audio is passed through an AIC (analog interface circuit) A/D to the McBSP which receives one sample at a time. When the receive register is full, it tells the EDMA to copy one 16-bit sample to a receive buffer. When that buffer is full, the EDMA interrupts the CPU and posts a TSK to run. This TSK processes (copies) the buffer to the transmit buffer. When that transmit buffer is full, the EDMA copies one 16-bit sample at a time to the Xmt McBSP and out through the D/A to your headphones.

You're Done

8 Lab 3: Using Client.pjt "Out of the Box"



8.1 Lab Overview:

- The goal of this lab is basically just to RUN the client.pjt example and observe what it can do "out of the box". This is the main example you might be modifying if you choose to use the NDK in your development. Lab5 walks you through the steps of modifying client.pjt to perform one service (in our example, just a DAEMON echo server).
- Please note that this lab uses the EVALUATION version of the NDK – version 1.92. If you have purchased the NDK, you will need to pay close attention to the paths used in this lab – your paths are most likely different than what are used in the procedures to follow.

8.2 Open Client.pjt and Run It

1. Set the Installation Environment variable.

Setting the following environment variable is a critical first step in using the NDK. This is one of the top stumbling blocks for first-time users. None of the examples that ship with the NDK will work unless this environment variable is set on the PC. This issue is documented, but we wanted to make sure we highlighted this in the Getting Started Guide.

Open up the PC's Control Panel and double-click on "System". Click on the "Advanced" tab. Near the bottom, click on the "Environment Variables" button. In the bottom window, check to see if the variable "NDK_INSTALL_DIR" is set to anything. If so, leave it alone. If not, click the "New" button and type in the following:

Variable name: `NDK_INSTALL_DIR`

Variable value: `C:\CCStudio_v3.3\ndk_1_92_eval\`

Click OK and close the control panel. Now, your examples that you paid $5000 for will actually work. By the way, we are using the evaluation version of the NDK. If these files are copied, they will work just fine. However, if used in a real application, the stack will reset itself if it runs for 24 hours straight (can you say time bomb?). If you are using a newer version of the NDK, please note that your directory structure might be slightly different.

Please note that if you purchased the NDK or have a different version of the NDK, the paths used in this lab will be different. Just substitute your actual path for the paths listed in the steps below.

---

2. Connect C6455 DSK to the open Ethernet connection on the desktop or laptop.

Connect your DSK to an open Ethernet connection on the laptop or desktop student machine. The open connection IP's address is set statically to 192.168.1.39. Desktop PCs can use a standard Ethernet cable (usually). Laptops require a crossover cable (usually).

3. Examine the directory structure of the NDK.

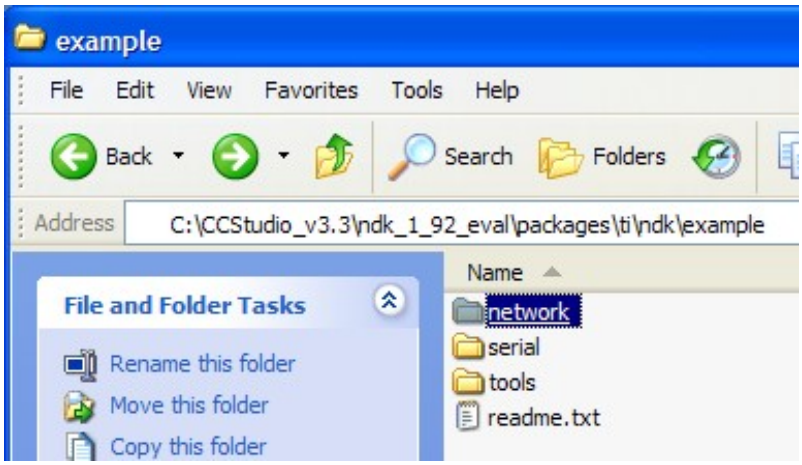
Remember, there are two downloads for the NDK: (1) platform-independent NDK package that contains the libraries, stack, main files, servers, nettools, winapps; (2) the network support package that is platform-DEPENDENT and contains the hardware specific libraries and example code. In our case, we're using the support package based on the C6455 DSK.

Using Windows Explorer, go to the following directory:

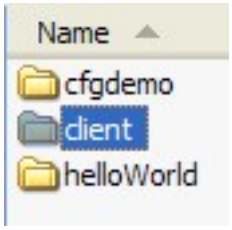
C:\CCStudio_v3.3\ndk_1_92_eval\packages\ti\ndk\example

Here you will find the example code for the NDK. There is a "common" set of files and a "board specific" set of files for each example.

You should see the following under the *example* directory:



So, you can see the different examples available. Client.pjt, the example we plan to use is located in the network folder. Open the *network* folder. You should see the following:



Under *client,* you can observe two directories: (1) *common* folder – which contains the common files loaded by the platform-independent NDK download; (2) *dsk6455* folder which contains the platform-dependent files (again, in our case, it is the dsk6455). You will find the project – client.pjt – in the dsk6455 directory.

4. **Make a copy of the "example" folder.**

   One of the other common mistakes that users make is that they modify code examples and then want to go back to the original – which is typically overwritten during development. One way to avoid major headaches is to make a copy of the example folder. So, let's practice. In Windows Explorer, go to the following directory:

   ```
   C:\CCStudio_v3.3\ndk_1_92_eval\packages\ti\ndk
   ```

   Do you see the folder "example"? In Windows, just drag and drop of copy of this folder AT THE SAME LEVEL (beneath \ti\ndk). Name the copy of this folder "EXAMPLE_COPY". Again, just in case you make a mistake modifying an NDK file, you can easily replace the original NDK file with your copy. It might (will) come in handy later.

Yes, we made a copy of the \example folder. However, to preserve the path names contained in the original NDK project (client.pjt), we will actually use the ORIGINAL files and paths for the remainder of this lab. The copy is only there in case something goes wrong. To say it again, when you open client.pjt a few steps later, use this path (NOT your copy path):
..\ndk_1_92_eval\packages\ti\ndk\example\network\client\dsk6455\client.pjt

---

5. **Close everything, close CCS, power cycle the DSK and open the project.**

   Close all existing projects. Close CCS. Power-cycle the DSK. Open CCS. Open the project client.pjt under the directory path (this is the original path – NOT your directory copy):

   ```
   C:\CCStudio_v3.3\ndk_1_92_eval\packages\ti\ndk\example\network\client\dsk6455
   ```

6. **Modify client.c to set static IP address for the DSK.**

   Locate the source file client.c and open it. Client.c defaults to using DHCP for configuring the IP addresses. We want to use a static IP address for the DSK, so we need to modify the code:

   Modify line 65 to read: *LocalIPAddr = “192.168.1.41”;

   Modify line 66 to read: *LocalIPMask = “255.255.255.0”;

   Save client.c. (Note: ensure that 192.168.1.41 does not conflict with any other local resources).

7. **Build, load, run the NDK.**

   Build and load the client project using the Rebuild All button: 

   This build will take about a minute. Why? Because everything plus the kitchen sink is in client.pjt – telnet service, DAEMON servers, HTTP service, etc. It’s all there right out of the box. We don’t know how it all works yet, but we first need to play with it, then dive into the details. Please note that although this project is called “client.pjt”, it is not just a client – it contains all kinds of services (FTP, HTTP, telnet) and servers (data, echo, DAEMON, etc.).

   Click Run.

   The CCS Stdout window will show the service status of the IP address assigned to the DSK as, “Network Added: If=1:192.168.1.41” (Note: If you modified the code in the step above, this is EXACTLY what your Stdout window should look like except for maybe the MAC address…)

   CCS Stdout pane should display something like this:

```
TCP/IP Stack Example Client
Using MAC Address: 00-0e-99-ff-ff-ff
Network Added: If-1:192.168.1.41
Service Status: Telnet   : Enabled  :          : 000
Service Status: HTTP     : Enabled  :          : 000
Link Status: 100Mb/s Full Duplex on PHY 1
```

   Write down the IP address of your DSK below:

   _____ . _____ . _____ . _____

Note: After the application has been assigned an IP address, the StackTest() will attempt to receive data from the default IP address, which is the address assigned to the board.

8.3 Play With the NDK Services in Client.pjt

8. PING the DSK.

Pinging the DSK is the simplest way to make sure all of your connections are working properly. If the ping fails, you have a cable or hardware problem (or you're not running client.pjt on the DSK).

So, first, make sure client.pjt is RUNNING on the DSK.

Second, let's see if the DSK is alive and operational:

Open a Command Prompt window (DOS shell).

Change directory to:
`C:\CCStudio_v3.3\ndk_1_92_eval\packages\ti\ndk\winapps`

Type in: `ping 192.168.1.41`

If it works, you'll see a "reply" in the command window. If the request cannot connect to the target, STOP, you have a problem. Nothing else will work.

9. Observe a data transfer using SEND.

While in the command window, try:

```
send 192.168.1.41 10
```

It might take 5-10 seconds to start. Be patient. 10 is the number_between_updates. If the connection is slow, you can reduce this to zero (i.e. don't put any value there). You should see a screen that looks like this:

```
C:\CCStudio_v3.3\ndk_1_92_eval\packages\ti\ndk\winapps>send 192.168.1.41 500
10 Sending 8192 bytes...passed - 8192000 bytes/s
20 Sending 8192 bytes...passed - 8192000 bytes/s
30 Sending 8192 bytes...passed - 12288000 bytes/s
…
```

Hit <Esc> or Ctrl-Z to stop the send command. Ok, so we've ping'd the DSK and we've sent data to it. What else can we do?

10. Observe an echo command.

The following command will send data to the DSK and it will echo it back. Try:

```
echoc 192.168.1.41 10
```

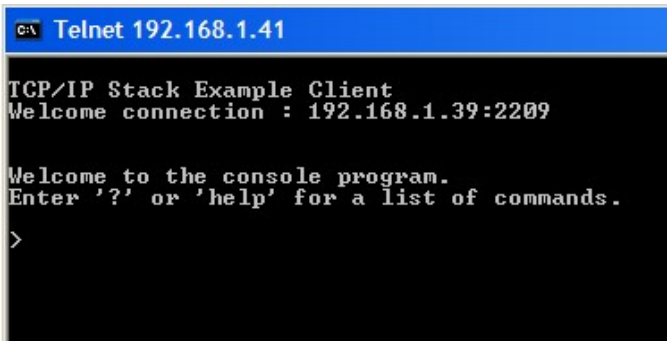
You should see the results in the command window (in fact, they will FLY by...). Use <Esc> to stop the echo. Wow, lot's of stuff is already working and we didn't program anything. We don't know a whole lot about WHAT is under the hood, but that's ok for now.

11. Run the telnet service.

Ok, two more “pokes” at the NDK software and we’re almost done with this lab. In the command window, type:

```
telnet 192.168.1.41
```

A window will open with the following information:



Type “?” and hit <Enter>. This will give you a list of commands.

Additional help for each command can be obtained by simply typing the command name, like “ping” our “route”. Others have no parameters (like “mem” and “log”).

Other commands can cause disconnection, like “quit” (closes the telnet connection), “reboot” (reboots the DSP) and “shutdown” (which restarts the NDK stack).

Type “quit” to close the telnet service.

12. Use the HTTP service.

Open your web browser.

Type in the board’s IP address: 192.168.1.41.

You will see the main web page stored in the NDK board and provided by the embedded web server (see picture).

You can obtain some statistics by either selecting an option and clicking the button “Display Selected Item” or clicking on one of the links at the end of the page. They differ by the method used to provide content: while the button “Display Selected Item” accesses a stored webpage in the server, the links at the end create each page dynamically using CGI scripts (Common Gateway Interface).

13. Look at the StackTest() function located in client.c.

Halt execution of client.pjt and open client.c. Find the StackTest() function (starting at about line 91).

We're going to study these functions more in-depth in the discussion material, but we wanted you to at least observe them for now. The next section (which is optional and FYI) contains a more detailed explanation that you can read later.

Locate the following function calls in StackTest():

```
rc = NC_SystemOpen( NC_PRIORITY_LOW, NC_OPMODE_INTERRUPT );

hCfg = CfgNew();

CfgAddEntry(. . .);

rc = NC_NetStart(hCfg, NetworkOpen, NetworkClose, NetworkIPAddr );
```

14. Let's look at some of the NDK documents:

In Windows Explorer, find the following folder:

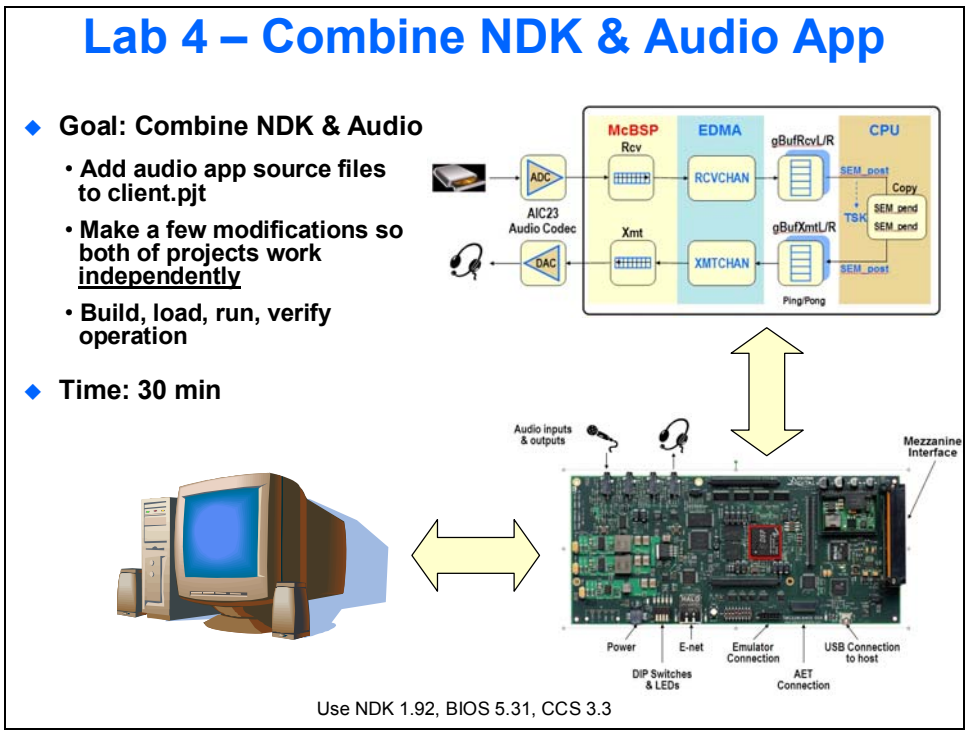
C:\CCStudio_v3.3\ndk_1_92_eval\packages\ti\ndk\docs\dsk6455

Open `Readme.htm`.

This contains some software installation instructions for the network support package for the 6455 DSK. If you go back under \docs, click on the stack folder. There are two .htm files in there that contain lots of helpful information about the NDK.

You're Done

9 Lab 4 – Combining the NDK and Audio Lab



9.1 Lab Overview:

The goal of this lab is to combine an NDK example (client.pjt) and the audio application from lab6B. The basic steps for combining an existing application into the NDK environment are basically the same for any application. We picked lab6B because the audio sounds the best (no filter). Some of the entire process has been done for you and placed in a “starter” directory. These steps are documented and you will have a chance to analyze what was done prior to going through the rest of the procedure yourself.

The reason this is a good project to attempt is to determine what resources and files might conflict when you add the two projects together. We will walk you through each change and why they need to be made.

9.2 Introduction

1. How to integrate an application with the NDK.

There are two methods for integrating an existing application with the NDK: (1) add your application files to the NDK environment; (2) add the NDK environment and files to your system.

While both can be accomplished, it is easier to add your application to the NDK's environment for one main reason: you know your application better than the NDK – so if you forget some hook or task or file, it is easier to catch if it is an application file vs. some obscure NDK task that you forgot to add. So, in the future, save yourself some headaches and start with an NDK example, add your system files and modify from there.

In a nutshell, when you combine two projects (yours and the NDK's), the following needs to occur:

- add all application source files to the NDK example project (e.g. client.pjt)
- port all DSP/BIOS settings from your application .tcf file to the NDK's .tcf file (dsk6455.tcf)
- change project options to include any CSL libraries used in your application
- modify your applications source files to use new headers
- remove any conflicts that occur (this is a big variable – could be interrupts, cache, system resources, memory usage, etc.)

We plan to go through each of these steps in this lab. Some have already been done for you, but some you will need to do yourself.

9.3 Modify Client.pjt to Include Application Files/Libraries

2. Close Code Composer Studio (CCS) and power cycle the board.

3. Copy audio files to same directory as client.pjt.

Our "application" is an audio pass-through built using DSP/BIOS tasks, EDMA, McBSP and a hardware interrupt. This code is the solution for lab6B of the IW6455 4-day workshop with one modification (explained later).

Open two Windows Explorer windows – one each for the following folders:

```
C:\CCStudio_v3.3\ndk_1_92_eval\packages\ti\ndk\example\network\client\dsk6455
C:\IW64x+\labs\starter\NDK_audio_starter
```

Copy the audio application files from the \NDK_audio_starter directory to the directory containing client.pjt. Overwrite the file **dsk6455.tcf**. This is the textual configuration file (tcf) that has been modified slightly to speed up this lab. You'll have a chance to view these changes in a few minutes.

4. **Open CCS and add the application files to client.pjt.**

   Open CCS. Open the project client.pjt located in the following directory:

   `C:\CCStudio_v3.3\ndk_1_92_eval\packages\ti\ndk\example\network\client\dsk6455`

   Select:

   Project → Add Files to Project…

   and add the following application source files to client.pjt (you can ctrl-click each one and add them all at the same time):

   - edma.c
   - edma_int_dispatcher.c
   - main.c
   - mcbsp_aic23.c
   - scr_priority.c
   - tasks.c

5. **Add CSL and BSL library files (for audio app) to the project**

   The audio pass-through lab required two library files (a Chip Support Library – CSL – file and a Board Support Library – BSL – file). Right-click on the Libraries folder in the project and select Add Files to Project. Browse and add the following library files:

   - `C:\IW64x+\csl_6455_v3.00.10.02\csl_c6455\lib\csl_C6455.lib`
   - `C:\CCStudio_v3.3\boards\dsk6455_v2\lib\dsk6455bsl.lib`

   When you merge your own application with the NDK, you'll need to add the libraries used in your application to the example NDK project.

6. **Add search paths for library include files**

   The libraries need include files. We need to add these paths to the "include search path". To add two new search paths, select:

   Project → Build Options → Compiler Tab → Preprocessor Category → Include Search Path

   At the end of the existing paths, type a semicolon ";" and then add the following two paths with a semicolon between them:

   `C:\IW64x+\csl_6455_v3.00.10.02\csl_c6455\inc;`

   `C:\CCStudio_v3.3\boards\dsk6455_v2\include`

   Click OK to save the changes.

7. **Remove the source file dsk6455.c**

   This source file that comes with the NDK conflicts with the BSL library from Spectrum Digital. Both of them are not needed. So, if your application uses a board support library, add the library you've been using and remove the one provided by the NDK.

   Right click on the source file **dsk6455.c** and select *Remove From Project.*

9.4 Modify Source Files

There are 3 main areas we need to focus on: (1) replacing the #include for the header file automatically generated by CCS (the NDK uses one name, your application uses a different one); (2) we currently have two main() routines – one has to be eliminated; (3) a conflict may exist regarding the hardware interrupt used by the NDK and the application.

8. Make changes to #include in application source files.

In three files – main.c, edma.c, tasks.c – you can find a #include at the top of the file. Open the source file main.c and look at the first #include:

```
#include "dsk6455cfg.h"
```

All three application source files noted above used the following #include statement:

```
#include "audio_appcfg.h"
```

This is because the name of the .tcf file for the audio application was “audio_app.tcf”. This header file is automatically generated by CCS and named after the .tcf file. Now that we’re using the NDK’s .tcf file (dsk6455.tcf), we needed to change the #include statements in our application source files. All three files have already been changed for you. The point here was to observe it and note that this change is necessary and why.

9. Erase one of the main() functions.

Our application had a main() function and so does the NDK. Because both our application and the NDK require DSP/BIOS, both main() functions return to the DSP/BIOS scheduler. In fact, the NDK’s main() is empty. We can’t have two, so let’s delete the NDK’s main() function.

In the source file **client.c** (which contains the stack startup and configuration of the NDK services), locate main() – about line 76. Remove the entire main() function from client.c.

Verify that the IP address and mask values are correct in client.c (these should have been set properly in a previous lab – just double-checking):

Ensure line 65 reads: *LocalIPAddr = “192.168.1.41”;

Ensure line 66 reads: *LocalIPMask = “255.255.255.0”;

Save client.c. and close.

10. Remove hardware interrupt conflict.

Just by chance, the audio application author used hardware interrupt #5 for the EDMA interrupt to the CPU (there are 12 available CPU interrupts numbered 4-15) which conflicts with the NDK's EMAC interrupt. To the author's knowledge (which is very suspect anyways), there is no documentation of which interrupt the NDK/EMAC uses. So, after debugging the problem, the author discovered that the NDK and audio application used the SAME interrupt (is that called Murphy's law or just plain bad luck?).

So, be forewarned – the NDK uses CPU interrupt #5 for the EMAC interrupt to the CPU. This is actually set in an NDK source file, so it could be changed. However, we'll leave the NDK files alone and change our application to use interrupt #6 instead.

Let's see where the NDK sets this interrupt. Open the NDK source file **dsk6455init.c**. Find the function: C6455EMAC_getConfig() – about line 62. Observe the following line of code:

```
*pIntVector = 5;
```

It is not intuitively obvious and this one fact alone may save you a day or two of pulling your hair out (the author of this lab is now bald you know). This value "5" in the code above is used in the EMAC driver to "plug" the interrupt vector table with the name of the EMAC's ISR function. This is a dynamic way of configuring interrupts.

Close this file without making any changes.

Our audio application uses a static way of configuring interrupts inside the .tcf file. Let's take a look at which interrupt it currently uses.

In the project window, expand the folder *DSP/BIOS Config*. Open the file **dsk6455.tcf**. Click on the + next to *Scheduling* and next to *HWI*. You can now see the list of CPU interrupts. Right-click on **HWI_INT5** and select *Properties*. This is exactly what the audio application hardware interrupt looks like. We know now that this resource (HWI_INT5) conflicts with the EMAC interrupt. We need to copy these properties to interrupt #6 (or any interrupt other than 5) to avoid the conflict.

Go to the next page…

Note the settings for HWI_INT5:

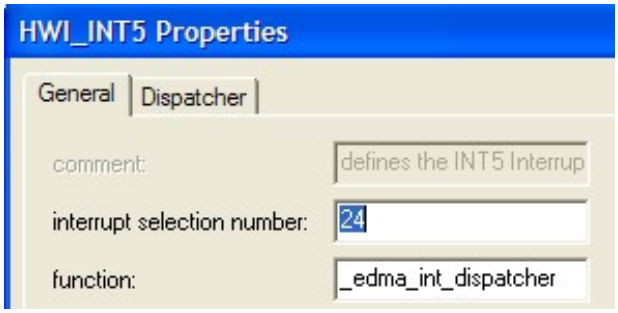
interrupt selection number: **24** (this is for the EDMA interrupt to the CPU)

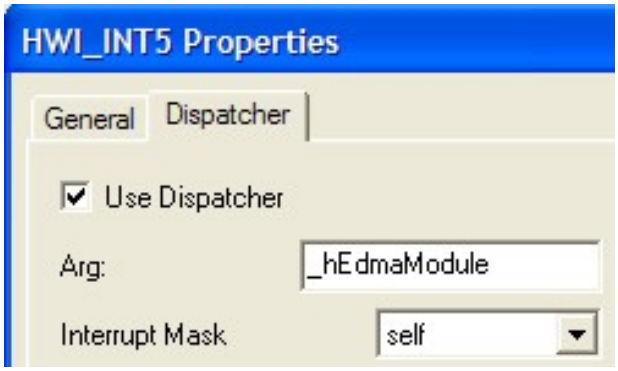
function: **_edma_int_dispatcher** (this checks to see which EMDA channel caused the interrupt)

Dispatcher: **check box "Use Dispatcher"**

Dispatcher: **Arg = _hEdmaModule**

See the pics below:





First, with the HWI_INT5 properties open, clear all settings so we don't confuse the compiler. For HWI_INT5, type in the "function" as HWI_unused. Also, click on the Dispatcher tab and make sure "Use Dispatcher" is unchecked.

Close HWI_INT5.

Right-click on HWI_INT6 and select Properties. Modify the properties to use the same settings as the pictures above.

Click OK and save the .tcf file.

In main.c, about line 64, there is a line of code that enables individual interrupts (int #5):

```
C64_enableIER(C64_EINT5);
```

Change the value to `C64_EINT6`. This will enable interrupt #6. You have now changed the application to use interrupt #6 instead of #5. Close and save main.c.

9.5 Port DSP/BIOS Settings From Application to NDK

The audio application makes extensive use of DSP/BIOS. All of these settings are done in the application's .tcf file. Because we can only have one .tcf file, we must port all the settings from the audio application into the NDK's .tcf file – **dsk6455.tcf**.

All of these changes have been done for you in the starter .tcf file, so the goal here is to observe the modifications and understand their implications.

We will go right down the list in the .tcf file in order and examine what was ported.

11. In the project window, open dsk6455.tcf.

12. Under System Settings, observe the cache and memory settings.

Click on the + next to *System*. Right-click on *Global Settings* and select *Properties*. Click on the 64PLUS tab. Notice that L2 cache is set to 256K and the DDR2 memory area is set as cacheable (MAR 224-255 is set to 0x0000FFFF). The NDK requires some L2 cache, so we simply maxed it. You could use less if necessary – see the NDK User Guide for more details on this.

Click Cancel to close the Properties window. Then click on the + next to *MEM*. Right-click on *IRAM* and select *Properties*. Notice the IRAM length change to 0x001C0000 (vs. 0x00200000 which is standard). This was done to make room for the 256K of L2 cache.

13. Observe Instrumentation settings.

Click on the + next to Instrumentation. If you had any LOG or STS objects in your application, you would need to port them over. LOG is used, for example, for LOG_printf() statements; STS are used for benchmarking areas of code. Our audio application used neither, so no porting was necessary.

14. Observe Scheduling settings.

Click on the + next to *Scheduling*. Click on the + next to *PRD*. Right click on the PRD object **PRD_waitPRD** and select properties. The audio application uses a periodic function to toggle an LED on the DSK. The object **PRD_waitPRD** is used to toggle the LED every 500ms. Just keep in mind that whatever settings your current application uses, they must be ported to the NDK's .tcf file.

Click on the + next to *TSK*. The audio application used two TSKs: (1) **TSK_DipLED** to toggle the LED; (2) **TSK_processBuffer** to process the audio samples (copy them from the receive buffer to the transmit buffer). TSK priorities can be an issue with the NDK. Generally, you can leave your priorities the same when you port your application to the NDK. However, if something is not working properly, it might be an issue of priority.

Click on the + next to *Synchronization*. Click on the + next to *SEM*. The audio application used 3 semaphores: (1) **rcvBuffReady**; (2) **xmtBuffReady**; (3) **waitPRD**. The rcv/xmt semaphores are for signaling the Scheduler when the EDMA has finished receiving or transmitting a buffer of audio data. waitPRD is used to signal the TSK to toggle the LED.

9.6 Build, Load and RUN !

Well, it's show time. Let's see if we have any build errors. If so, we'll fix 'em and then run the project to see if both applications work side by side.

15. Rebuild All.

Click the Rebuild All button. Again, it will take a while because we have every NDK service running and enabled. In the optional lab that follows, it takes you through the steps to trim the NDK down to a single service.

Did you get any errors? If so, you'll need to fix them. If you struggle for more than 5 minutes, ask your instructor for help.

If you have the "Load Program After Build" option set, the .out file should have loaded. If not, select:

File → Load Program

and load the file: \dsk6455\bin\client.out. If your .out file does not load, always check to make sure you are connected to the board (look in the lower LH corner of the CCS window – does it say "Connected" or "Not Connected"?). Use <ALT-C> to connect if needed.

16. Get some music playing.

There should be a shortcut to some .mp3 files on the desktop. Click on one of the songs which will open up Windows Media Player. Make sure that Play → Repeat is on (so the music just doesn't stop on you). If there is not shortcut on the desktop, use Windows Explorer, navigate to C:\IW64x+\music and click on a song.

17. Run the code.

Click the *Run* button. You should see the STDOUT screen show a proper link status (as in the previous lab).

Make sure your headphones are connected – do you hear the music playing? If so, the audio code is running fine. If not, there is a mistake somewhere.

If the audio is running fine, open a DOS command window and navigate to:

```
C:\CCStudio_v3.3\ndk_1_92_eval\packages\ti\ndk\winapps
```

Type in:

```
echoc 192.168.1.41  10
```

Do you see data being echoed back to the PC? If not, something is amiss. If so, you have successfully integrated an application with the NDK and they are both running side by side. Press ctrl-Z or <ESC> to stop the echo. Close the command window.

You're Done

10 Modifying Client.pjt

When you purchase the NDK, the best way to get started is to use one of the examples provided – namely client.pjt. This section provides a step by step procedure for modifying the contents of client.pjt. In this example, we chose to trim the NDK to only perform a DAEMON echo server. In your application, you might choose just to leave the HTTP service or telnet. The practice of modifying client.pjt will be similar.

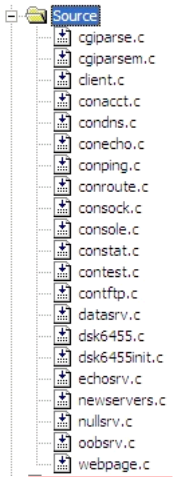
This will help you understand which source files and libraries are absolutely essential and which are not.

10.1 Deleting Client.pjt Source Files

Shown below, we plan to remove the telnet and HTTP services and some sockets programming server files.

Client.pjt – Console, html, servers

- In this example, we are not using sockets programming (we will later). There is a built-in DAEMON server that will provide our simple echo.
- So, there are many services (source files and code) that we can delete.
- Let's take this step by step…



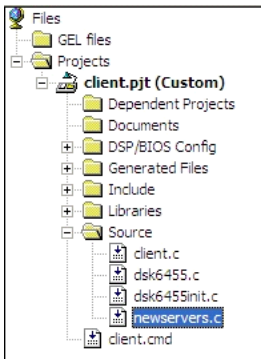
1. Remove console source files
   - user prompt when running telnet service
   - remove: console.c, con???.c
2. Remove html content-related files
   - html, web page service
   - remove: webpage.c, cgiparse.c, cgiparsem.c,
3. Remove server files
   - echosrv.c, datasrv.c, nullsrv.c, oobsrv.c,
   - These are written using sockets programming instead of DAEMON interface

10.2 Client.pjt – Modifying Newservers.c

In newservers.c file, we'll remove the unused servers and leave the echo server.

Client.pjt – Modify newservers.c

- To perform a simple echo (using the DAEMON server), we no longer need the following servers: data, null, oob (in file newservers.c)



4 Remove the unused servers
- dtask_tcp_datasrv
- dtask_tcp_nullsrv
- dtask_tcp_oobsrv

```
int dtask_tcp_datasrv( SOCKET s, UINT32 unused )
{
    struct timeval to;
    int            i,size;

    (void)unused;

    /// Configure our socket timeout to be 5 seconds
    to.tv_sec  = 5;
    to.tv_usec = 0;
    setsockopt( s, SOL_SOCKET, SO_SNDTIMEO, &to, sizeof( to ) );
    setsockopt( s, SOL_SOCKET, SO_RCVTIMEO, &to, sizeof( to ) );
```

Note: you can delete everything from the _datasrv function shown above all the way to the end of the code in newservers.c.

Next, we'll modify client.c. But first, let's look at the client.c source code…

10.3 Client.c Source File – Overview

Client.c Source file - Overview

- StackTest() is the main and most important function in client.c. It is responsible for configuring and booting up the stack
- NetworkOpen/Close initialize/kill the user-defined applications (TSKs)
- NetworkIPAddr, CheckDHCPOptions are callback functions to ensure connectivity

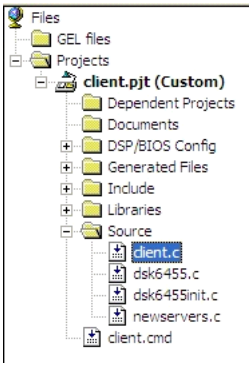
Next, we'll modify client.c to delete code that is not used for the simple echo example…

```
main() {}
StackTest() {    // Boots up the stack and runs forever
    • Prepares for system startup (NC_SystemOpen)
    • Creates a new network configuration (CfgNew)
    • Assigns local IP Address
    • Initializes services (http, telnet)
    • Fires up the stack (NC_NetStart)
}
NetworkOpen() {
   // initializes the user-defined applications (TSKs)
}
NetworkClose() {
   // kills the user-defined applications (TSKs)
}
NetworkIPAddr() {
  // call back function from the ethernet driver
}
CheckDHCPOptions() {
  // call back function from the services init routines
}
```

10.4 Client.c – Remove Telnet Service

Client.c – Remove Telnet Service

- We have already removed the services such as telnet, http, etc.
- Therefore, we need to remove the calls to these services in client.c
- In StackTest(), let's remove the telnet code first…



5 Remove call to telnet service in client.c

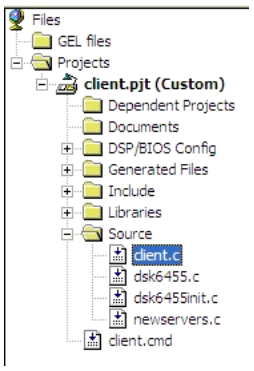
- Remove config handler to the telnet service:
  `CI_SERVICE_TELNET telnet;`
- Remove config code starting at (this line)…
  `bzero(&telnet, sizeof (telnet));`
- …to this line (6 lines of code, see example below)
  `CfgAddEntry (hCfg, CFGTAG_SERVICE, ..0);`

```
// Specify TELNET service for our Console example
bzero( &telnet, sizeof(telnet) );
telnet.cisargs.IPAddr  = INADDR_ANY;
telnet.cisargs.pCbSrv  = &ServiceReport;
telnet.param.MaxCon    = 2;
telnet.param.Callback  = &ConsoleOpen;
CfgAddEntry( hCfg, CFGTAG_SERVICE, CFGITEM_SERVICE_TELNET, 0,
             sizeof(telnet), (UINT8 *)&telnet, 0 );
```

10.5 Client.c – Remove HTTP Service

Client.c – Remove HTTP Service

- Now, we can remove the http configuration steps and the calls to removed functions in StackTest()....



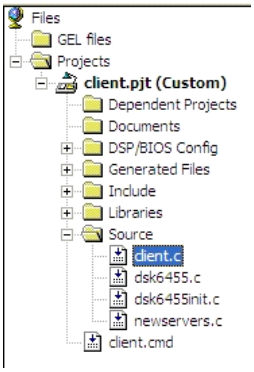
6 Remove http service in client.c

- Remove config handler to the telnet service:
  `CI_SERVICE_HTTP http;`
- Remove file allocation system (to hold the webpages) – initialized by this function:
  `AddWebFiles();`
- Remove authentication system for webpages starting at the bracket BEFORE this line...
  `CI_ACCT CA;`
- ...to include the bracket AFTER this line:
  `CfgAddEntry (hCfg, CFGTAG_ACCT, …0);`
- Remove config code starting at this line...
  `bzero(&http, sizeof (http));`
- ...to include this line: (total of 4 lines of code):
  `CfgAddEntry (hCfg, CFGTAG_SERVICE, ..0);`

10.6 Client.c – Remove "USE_OLD_SERVERS"

Client.c – Remove "USE_OLD_SERVERS"

- For legacy purposes, there is a #define for USE_OLD_SERVERS. We can simply remove this entire #if / #else code.
- This causes confusion, because the original client.c has two versions of NetworkOpen()/Close() (because of this #if code). So, let's delete it:



7

- Remove #if USE_OLD_SERVERS code. Remove the ~50 lines of code from...
  `#if USE_OLD_SERVERS` (near line 270 or so)
- ...to (and including) the #else 50 lines further down
- Remove last #endif after the function:
  `NetworkClose();`
- Remove the following #define (near top of code):
  `#define USE_OLD_SERVERS 0;`

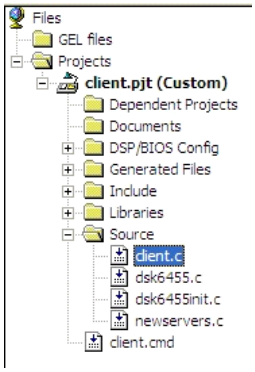
```
#if USE_OLD_SERVERS

///
/// System Task Code [ Traditional Servers ]
///
static HANDLE hEcho=0,hData=0,hNull=0,hOob=0;
                    •
                    •
                    •
///
```

10.7 Client.c – Remove Unused DAEMON Servers

Client.c – Remove Unused DAEMON Servers

- Remove the unused DAEMON servers (both the init and free calls)
- However, leave the DAEMON echo servers (that's what we want to use)
- These are found in NetworkOpen() and NetworkClose()…



8 Remove unused DAEMON servers (init, free)

- Remove init for hData, hNull, hOob:

```
hData = DaemonNew(SOCK_STREAM,..3);
hNull = DaemonNew(SOCK_STREAMNC,..3);
hOob = DaemonNew(SOC_STREAMNC,..3);
```

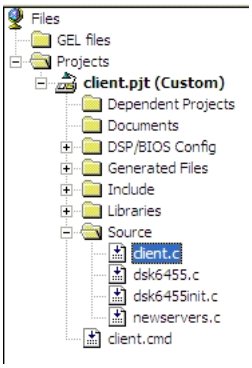
- Remove free for hData, hNull, hOob:

```
DaemonFree(hData);
DaemonFree(hNull);
DaemonFree(hOob);
```

10.8 Client.c – Clean Up

Client.c – Cleaning Up

- Clean up a few misc calls because we have removed some services
- Specifically: ConsoleClose(); RemoveWebFiles, and a few handles…



9 Cleanup some unused code

- Near the middle of StackTest(), remove TCP/UDP buffer/timing config options starting at the first line of code that says…

```
rc = 8192;
```

…to include this line:

```
#endif after CfgAddEntry(hCfg, CFGTAG_IP, …0);
```

- Remove:

```
RemoveWebFiles();
```
(near bottom of StackTest)

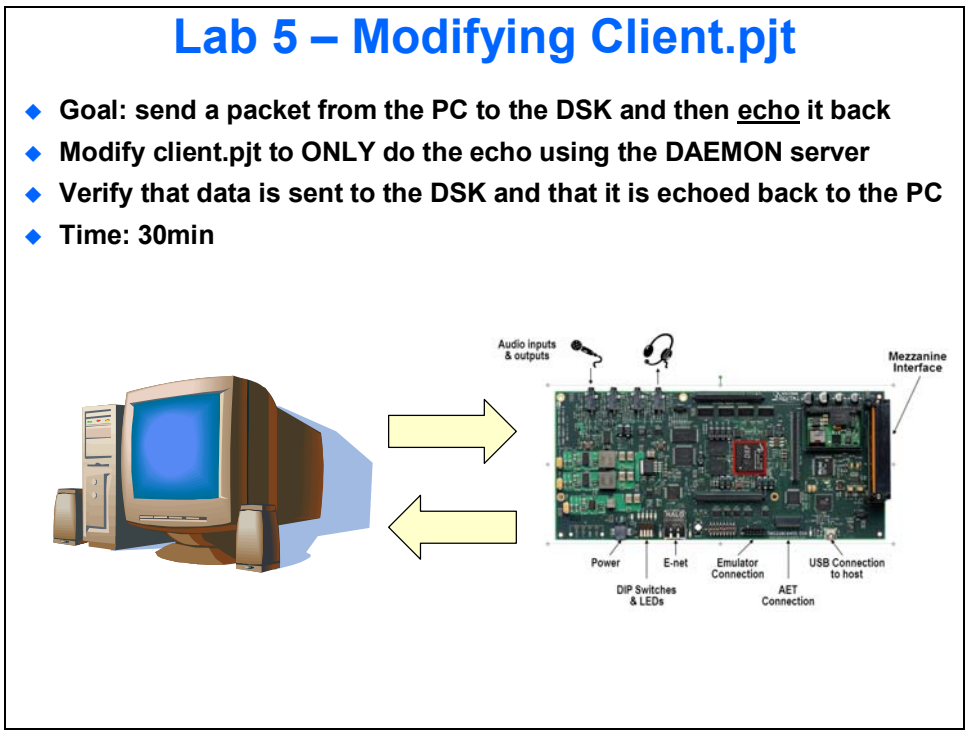
```
ConsoleClose();
```
(in NetworkClose())

- Remove handles that are not used (just above the NetworkOpen() fxn)

```
static HANDLE hData, hNull, hOob;
```

Note: Original .text size = 178KB
Stripped down = 120KB

11 Lab 5 – Modify Client.pjt



11.1 Lab Overview:

- Now that we have had a chance to play with client.pjt, it's time to learn how to MODIFY it to do only what we want it to do. Let's assume for the moment that we don't need the HTTP, console and telnet services. We want to simply have the DSK be a server (a DAEMON server) and echo back what we send it. Well, there is a ton of code we need to strip. What can we remove and what needs to stay? The answer lies ahead as you go through the steps….

- Keep in mind that this is ONLY an example. We arbitrarily chose to strip down client.pjt to only contain a DAEMON echo server. You might choose to only have the HTTP service running or a few of the services. Either way, this will give you a feel for what you can remove and how the client.pjt is built.

- The slides from the previous discussion are intentionally duplicated below so that you don't have to refer back in the document to view the lab steps. Plus, it adds a few pages that make the authors look like they created a really BIG document (more bonus money, more solid look/feel, etc.). By the way, what is "bonus" anyway? I used to know, but the definition slips my mind these days…

11.2 Modify Client.pjt

1. Introduction.

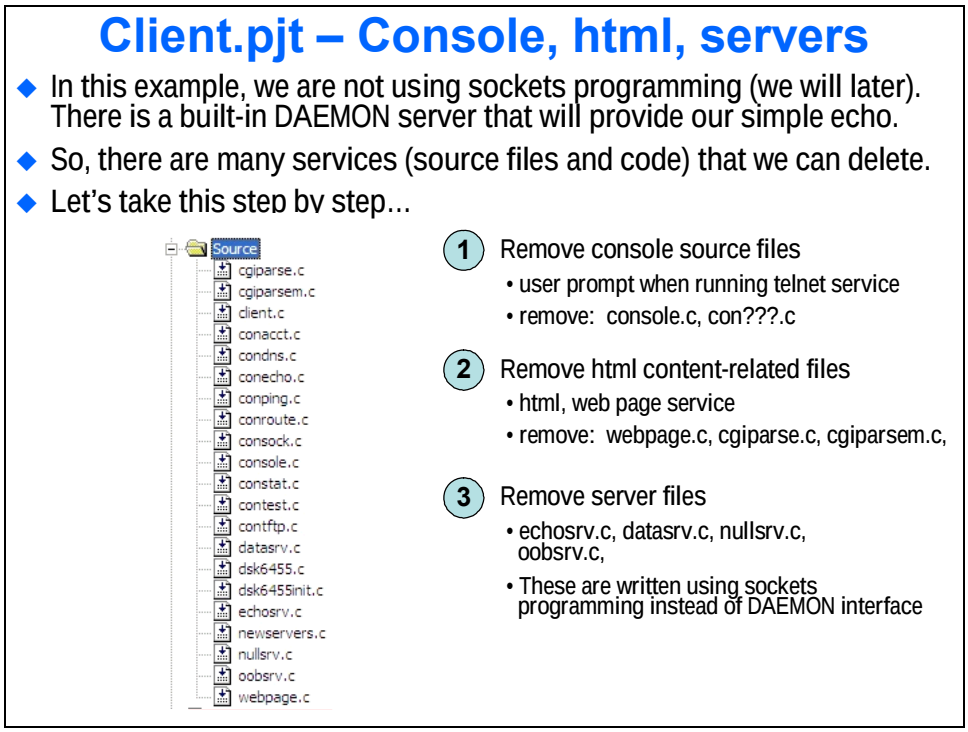
This lab will basically follow the discussion material exercise we covered regarding modifying client.pjt. We discussed what needs to be kept and why and how to delete unused services. The section was called “Client.pjt – your starterware”. It showed the 9 steps required to modify client.pjt to leave on the basic DAEMON server. Your job will be to follow those steps in order to come up with the next solution. Once you’re done, we’ll use Ethereal (now, replaced by Wireshark) to observe the results.

2. Open Client.pjt

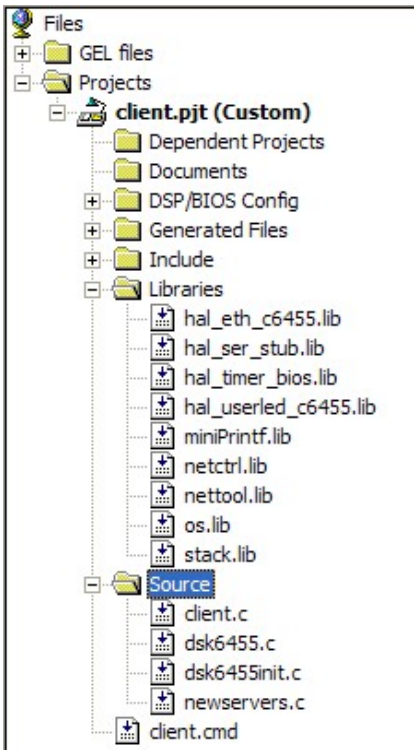
As before, open client.pjt. This may be your “combined” app+NDK or just the NDK. The procedure is the same. The slides shown in the material are duplicated here for your convenience. As a little tutorial, the following steps show some screen captures to check your work at specified points to make sure nothing is deleted that is needed. So, move on to the next step.

3. Steps 1-3 (remove console, html, server files)

Do steps 1-3 as follows:



When you are finished with steps 1-3, your project files should look like this (unless you have application files in there as well):

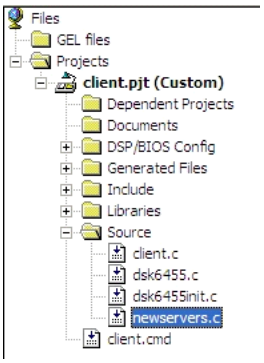


Double-check to make sure the Source files listed above are the only ones you have left (other than the maybe any extra application files). If you made a mistake, go back and add the necessary files back to the project. All libraries should already be in place – all are important and none can be deleted.

4. **Step 4 – remove the unused servers.**

Client.pjt – Modify newservers.c

- To perform a simple echo (using the DAEMON server), we no longer need the following servers: data, null, oob (in file newservers.c)



4 Remove the unused servers
- dtask_tcp_datasrv
- dtask_tcp_nullsrv
- dtask_tcp_oobsrv

```
int dtask_tcp_datasrv( SOCKET s, UINT32 unused )
{
    struct timeval to;
    int            i,size;

    (void)unused;

    // Configure our socket timeout to be 5 seconds
    to.tv_sec  = 5;
    to.tv_usec = 0;
    setsockopt( s, SOL_SOCKET, SO_SNDTIMEO, &to, sizeof( to ) );
    setsockopt( s, SOL_SOCKET, SO_RCVTIMEO, &to, sizeof( to ) );
```

Note: you can delete everything from the _datasrv function shown above all the way to the end of the code in newservers.c.

Next, we'll modify client.c. But first, let's look at the client.c source code…

Open the file newservers.c. You need to remove the datasrv, nullsrv and oobsrv. Remove the code for these servers. Each of these servers starts out with code that looks like this:

```
130: int dtask_tcp_datasrv( SOCKET s, UINT32 unused )
131: {
132:     struct timeval to;
133:     int            i,size;
134:
135:     (void)unused;
136:
137:     // Configure our socket timeout to be 5 seconds
138:     to.tv_sec  = 5;
139:     to.tv_usec = 0;
140:     setsockopt( s, SOL_SOCKET, SO_SNDTIMEO, &to, sizeof( to ) );
141:     setsockopt( s, SOL_SOCKET, SO_RCVTIMEO, &to, sizeof( to ) );
```

Remove the entire function for all 3 servers listed above. Basically, you start with the tcp_datasrv code shown above and delete everything to the end of the file. Save your changes.

5. **Step 5 – remove the telnet service and calls/configuration for it.**

Client.c Source file - Overview

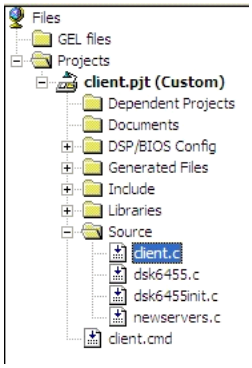
- StackTest() is the main and most important function in client.c.
  It is responsible for configuring and booting up the stack
- NetworkOpen/Close initialize/kill the user-defined applications (TSKs)
- NetworkIPAddr, CheckDHCPOptions are callback functions to ensure connectivity

Next, we'll modify client.c to delete code that is not used for the simple echo example…

```
main() {}
StackTest() {    // Boots up the stack and runs forever
    • Prepares for system startup (NC_SystemOpen)
    • Creates a new network configuration (CfgNew)
    • Assigns local IP Address
    • Initializes services (http, telnet)
    • Fires up the stack (NC_NetStart)
}
NetworkOpen() {
   // initializes the user-defined applications (TSKs)
}
NetworkClose() {
   // kills the user-defined applications (TSKs)
}
NetworkIPAddr() {
  // call back function from the ethernet driver
}
CheckDHCPOptions() {
  // call back function from the services init routines
}
```

Client.c – Remove Telnet Service

- We have already removed the services such as telnet, http, etc.
- Therefore, we need to remove the calls to these services in client.c
- In StackTest(), let's remove the telnet code first…



5 Remove call to telnet service in client.c

- Remove config handler to the telnet service:
  `CI_SERVICE_TELNET telnet;`
- Remove config code starting at (this line)…
  `bzero(&telnet, sizeof (telnet));`
- …to this line (6 lines of code, see example below)
  `CfgAddEntry (hCfg, CFGTAG_SERVICE, ..0);`

```
// Specify TELNET service for our Console example
bzero( &telnet, sizeof(telnet) );
telnet.cisargs.IPAddr = INADDR_ANY;
telnet.cisargs.pCbSrv = &ServiceReport;
telnet.param.MaxCon   = 2;
telnet.param.Callback = &ConsoleOpen;
CfgAddEntry( hCfg, CFGTAG_SERVICE, CFGITEM_SERVICE_TELNET, 0,
             sizeof(telnet), (UINT8 *)&telnet, 0 );
```

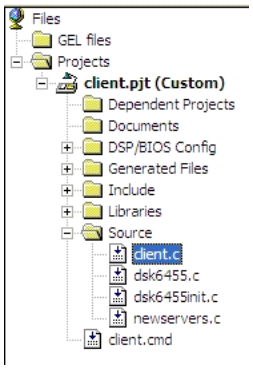
Open client.c. Find the StackTest() function. First, remove the call to the config handler for the telnet service (as show in the discussion material). Then, remove the config code that looks like this:

```
// Specify TELNET service for our Console example
bzero( &telnet, sizeof(telnet) );
telnet.cisargs.IPAddr = INADDR_ANY;
telnet.cisargs.pCbSrv = &ServiceReport;
telnet.param.MaxCon   = 2;
telnet.param.Callback = &ConsoleOpen;
CfgAddEntry( hCfg, CFGTAG_SERVICE, CFGITEM_SERVICE_TELNET, 0,
             sizeof(telnet), (UINT8 *)&telnet, 0 );
```

6. **Step 6 – remove the HTTP service and calls/configuration for it.**

Client.c – Remove HTTP Service

- Now, we can remove the http configuration steps and the calls to removed functions in StackTest()....

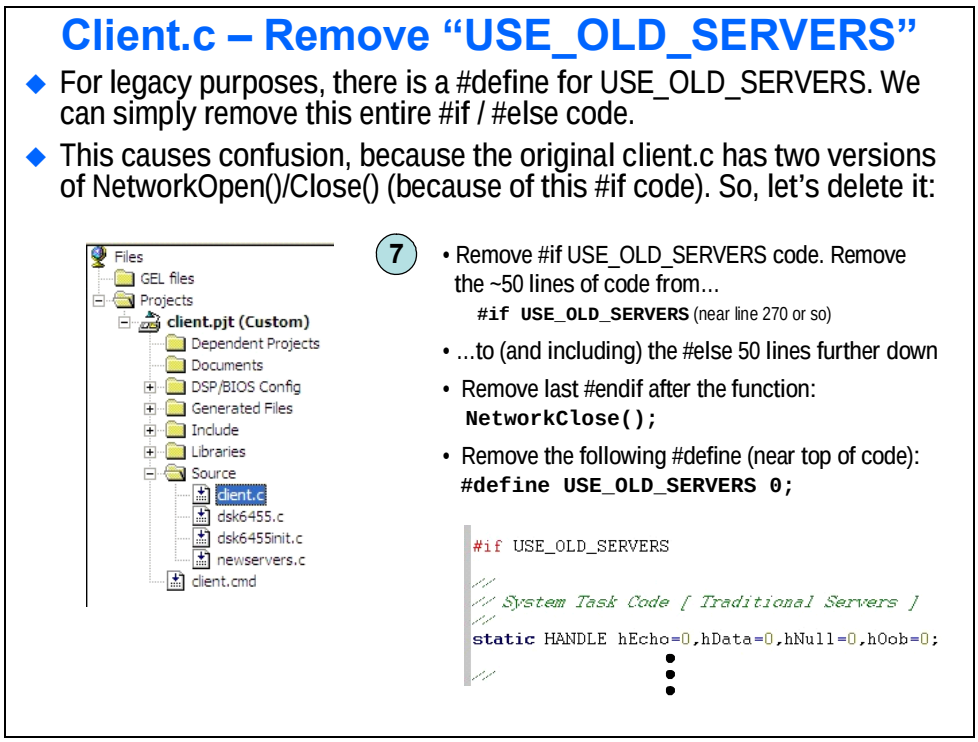


6 Remove http service in client.c

- Remove config handler to the telnet service:
  `CI_SERVICE_HTTP http;`
- Remove file allocation system (to hold the webpages) – initialized by this function:
  `AddWebFiles();`
- Remove authentication system for webpages starting at the bracket BEFORE this line...
  `CI_ACCT CA;`
- ...to include the bracket AFTER this line:
  `CfgAddEntry (hCfg, CFGTAG_ACCT, …0);`
- Remove config code starting at this line...
  `bzero(&http, sizeof (http));`
- ...to include this line: (total of 4 lines of code):
  `CfgAddEntry (hCfg, CFGTAG_SERVICE, ..0);`

In StackTest(), remove the HTTP code specified in the above material.

7. Step 7 – remove the #if USE_OLD_SERVERS code



This is legacy code that is not needed. There are currently TWO NetworkOpen() and NetworkClose() functions. This can get confusing. So, to avoid this confusion, let's remove the set that we are NOT using.

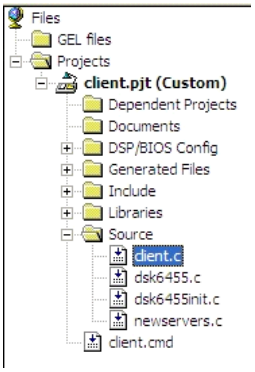
Locate the #if USE_OLD_SERVERS (near line 270 or so) and delete from this #if all the way down (about 50 lines) to (and including) the next #else. Delete. Then, scroll down just below the NetworkClose() function and delete the #endif.

Now, near the top of the code, remove the #define for USE_OLD_SERVERS. Save your changes.

8. **Step 8 – remove the unused DAEMON servers.**

Client.c – Remove Unused DAEMON Servers

- Remove the unused DAEMON servers (both the init and free calls)
- However, leave the DAEMON echo servers (that's what we want to use)
- These are found in NetworkOpen() and NetworkClose()...



8 Remove unused DAEMON servers (init, free)

- Remove init for hData, hNull, hOob:

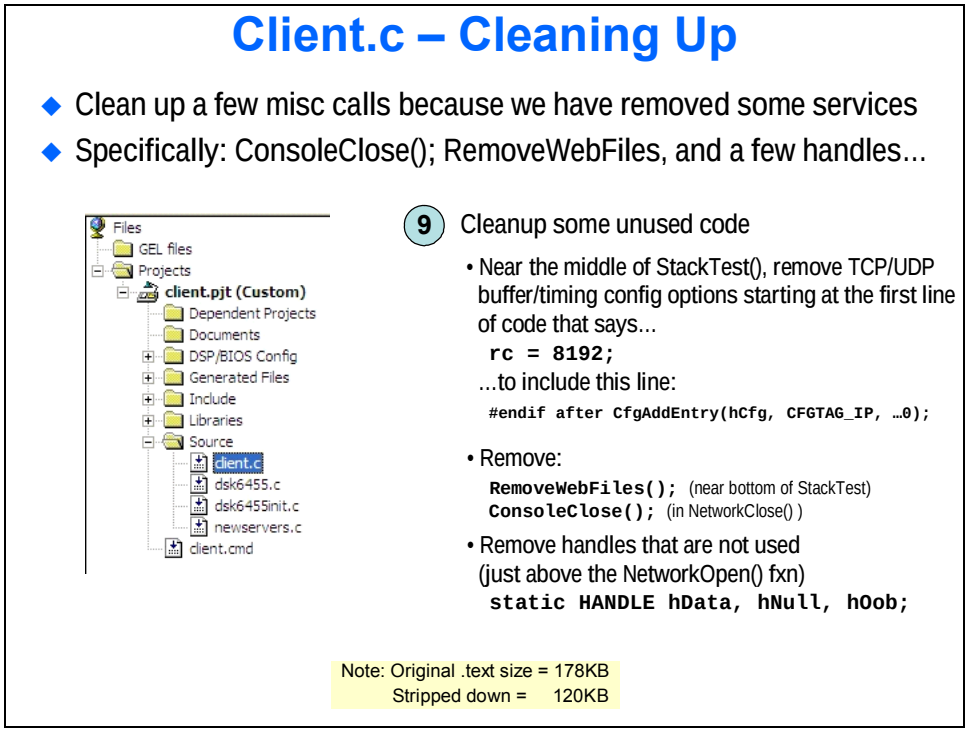
```
hData = DaemonNew(SOCK_STREAM,..3);
hNull = DaemonNew(SOCK_STREAMNC,..3);
hOob = DaemonNew(SOC_STREAMNC,..3);
```

- Remove free for hData, hNull, hOob:

```
DaemonFree(hData);
DaemonFree(hNull);
DaemonFree(hOob);
```

In client.c, locate the NetworkOpen() and NetworkClose() functions. In NetworkOpen() remove the DaemonNew() calls for the other servers (i.e. leave hEcho and hEchoUdp alone – delete the inits for the rest). In NetworkClose(), do the same for the DaemonFree() calls. Follow the directions for step 8 in the discussion material. Here, we are leaving the DAEMON echo servers which is what we plan to use for echoing data back from the DSK.

9. **Step 9 – cleanup unused code.**



In client.c, follow Step 9 from the discussion material and remove the unused code as specified. Save your changes.

10. **Build and load the new modified client.pjt.**

    Build and load and fix any errors that occurred. If there are warnings about variables that are unused, you didn't delete everything properly – go back and delete the init of those variables and do an incremental build. Once you have a clean build (and load), move on…

11. **Run Ethereal (actually, it's now called Wireshark).**

    Ethereal is a public domain newtwork analyzer (available for free at www.ethereal.com).

    Minimize CCS and locate the desktop icon for Ethereal. Double-click to run the program. Select Capture -> Interfaces. If more than one network interface is present on the machine, click the Capture button associated with the PC's IP address (192.168.1.39). Wait for 10 seconds and observe the network traffic flow. There shouldn't be any because we haven't run the code yet. There might be a few stray UDP packets – but no TCP packets. Leave the ethereal capture screen up on your display.

Update: as of 2Q08, Ethereal has been replaced by Wireshark – available for free at http://www.wireshark.org/. In these labs, you can actual use any network analyzer that you have (or can download).

---

12. Run Client.pjt.

Click the Run button in CCS to run the new client.pjt. The DSK will now be running a simple DAEMON echo server. We now need to send some data to the DSK and see if it echos back.

Open a command window. Resize this window so that you can see both the Ethereal (uh, Wireshark) capture screen AND the command window. In the command window, go to the \winapps directory and type:

```
echoc 192.168.1.41 10
```

You should immediately see the captured data show up in Ethereal (that's Wireshark now). After 5 seconds, click Stop and check the network traffic.

Ok, so we modified client.pjt to do just a simple DAEMON echo – not that difficult. In the next lab, we'll use socket programming APIs to send data from the DSK to the PC and use Wireshark (now that's better) to capture it.

13. Peruse the .tcf file.

How is this client.pjt system configured? What memory areas (external, internal) are allocated and why? Is cache turned on?

Open the .tcf file and observe the following: memory areas, where each section is allocated, scheduler – TSKs and PRDs, cache sizes, etc. Close the .tcf when done.

11.3 Save Your Solution and Close CCS

14. Save the following files using Windows Explorer:

- \example\network\client\dsk6455\bin\client.out (name it: client-lab5.out)
- \example\tools\common\servers\newservers.c (name it: newservers-lab5.c)
- \example\network\client\common\client.c (name it: client-lab5.c)
- Into the following directory:

```
C:\IW64x+\solutions_student\Lab5
```

You may need to create this directory using Windows Explorer and then copy your files.

Close CCS.

You're Done

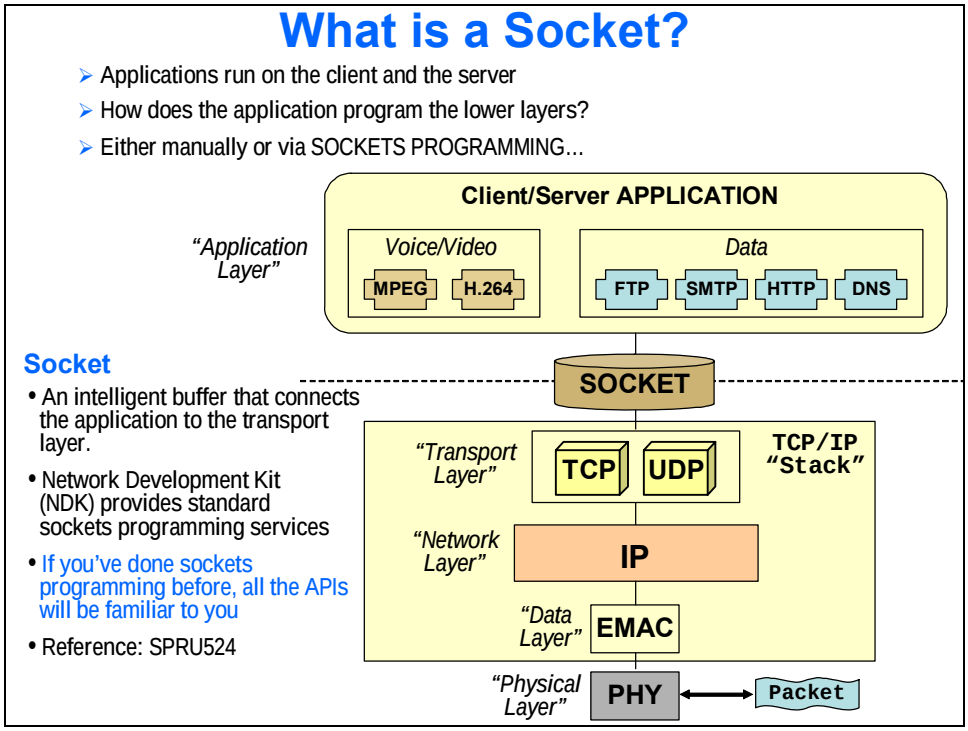
12 Sockets Programming

In this section, we will help you understand how sockets programming works. If you have done sockets programming in the past, this information will look very familiar to you. Typically, when you have an O/S like Unix or Linux, you have a kernel and a file system. In the NDK, we have a kernel – it is DSP/BIOS. However, BIOS doesn't have a file system. So, if you're familiar with sockets programming from an O/S that used a file system, you may want to pay particular attention to how we manage a file system in the NDK. Otherwise, the APIs and how sockets programming works is the same.

If you're new to sockets programming, this introduction will help, but will not suffice to get you up to speed completely. Further training on this topic is most likely needed.

12.1 What is a Socket?

A socket is the interface between the application code and the TCP/IP stack. Literally, it is an intelligent buffer that contains information regarding a live connection – for example, the FTP application is using port "x" at IP address "y" and is currently active. The socket also contains pointers to where the data is stored.



12.2 TCP – Sockets Programming APIs

Shown below is a basic TCP Sockets Programming Server/Client example. There are many more APIs – we're just showing the basic ones here.

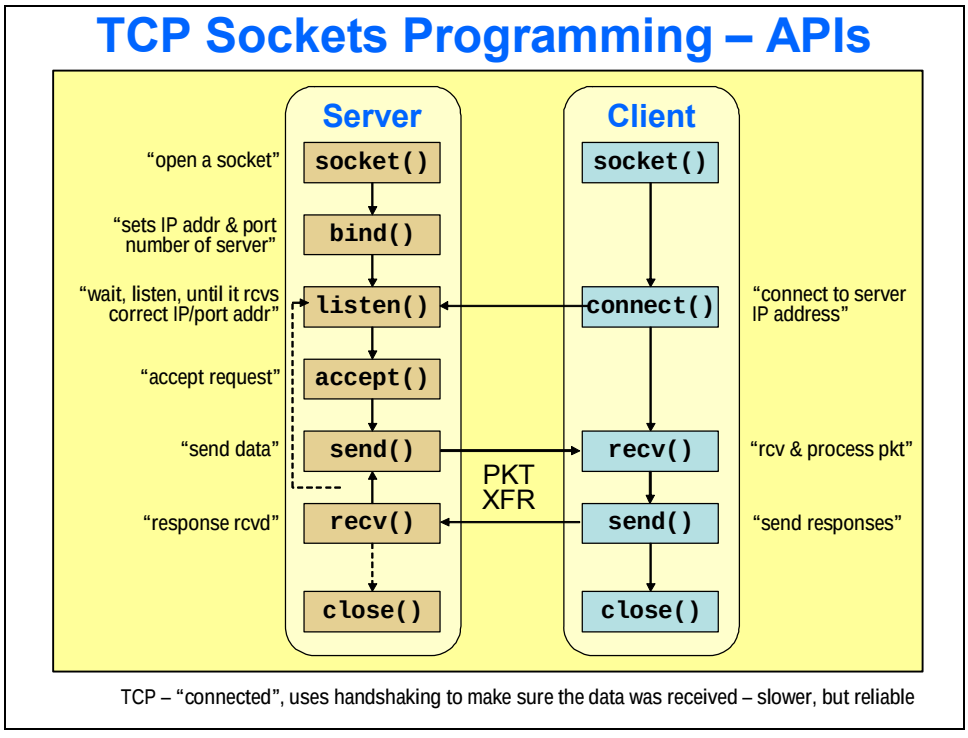
First, the Server opens a socket (for every server you need a different socket). Then it binds the socket to the IP address and the port number of the server. This really tells the world what kind of a server it is (for example maybe an HTTP or FTP etc). Then it just waits and listen until it receives something with the correct IP and port address.

On the client side, the client opens a socket (for example to try to get a web page). Then it connects to the server IP address and the port number of the server.

The server then accepts the request from the Client and sends the requested data to the Client. The client receives the packet and process the packet and sends a response to the Server that it got the packet. (TCP is a "connected" protocol and therefore has handshaking involved – much more reliable, but slower).

The server then receives the response so that it knows that the Client got the packet. If done, then close the socket – if not, go back to listen.

We'll see the actual code example for TCP shortly…

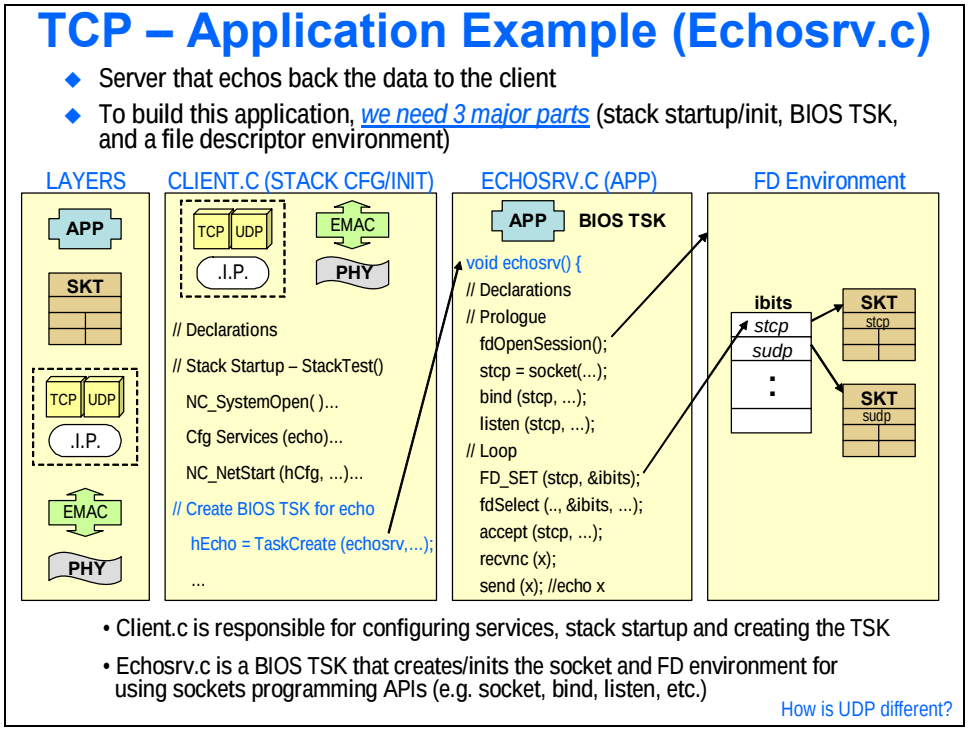


12.3 TCP – Application Example (echosrv.c)

The slide below provides the overall picture of how things link together and what is required to use sockets programming. The layers on the left are a shorthand view of the overall diagram a few slides ago (from app to the PHY). These “icons” are duplicated in the other sections to show which files modify these layers. Client.c is the example stack startup code that comes with the NDK. It is responsible mainly for opening the system, configuring the services and starting the stack. All of these functions are contained in the StackTest() routine (near the top of client.c). Two other main responsibilities are to create the necessary BIOS TSKs using NetworkOpen().

Echosrv.c is actually the application – this is where the BIOS TSK is programmed. The first item in any NDK TSK is fdOpenSession – this allocates the file descriptor (FD) environment that will be used throughout the TSK. Note that each TSK has a prologue (executed once), a “forever” loop (that can block on incoming data) and an epilogue (clean up code when session is closed). Then, in echosrv.c, you see the sockets programming APIs such as socket (create a socket), bind, listen, etc. These match the TCP flowchart shown previously.

Sockets programming requires a file descriptor environment to be created. fdOpenSession() allocates the environment. ibits is a file descriptor set that contains 16 pointers to sockets – of which two are shown in this diagram. The pointers are placed into the sets using the macro FD_SET. The sockets contain the name of the socket plus other parameters such as IP address, type of socket (TCP or UDP) and pointers to packet buffers. Sockets are allocated on the heap (so they must be freed when not in use) and the actual packet buffers are in memory configurable by the user. So, whenever you see “stcp”, this is “socket TCP” – just a name, but that’s what we intended it to represent.



12.4 File Descriptor (FD) Environment

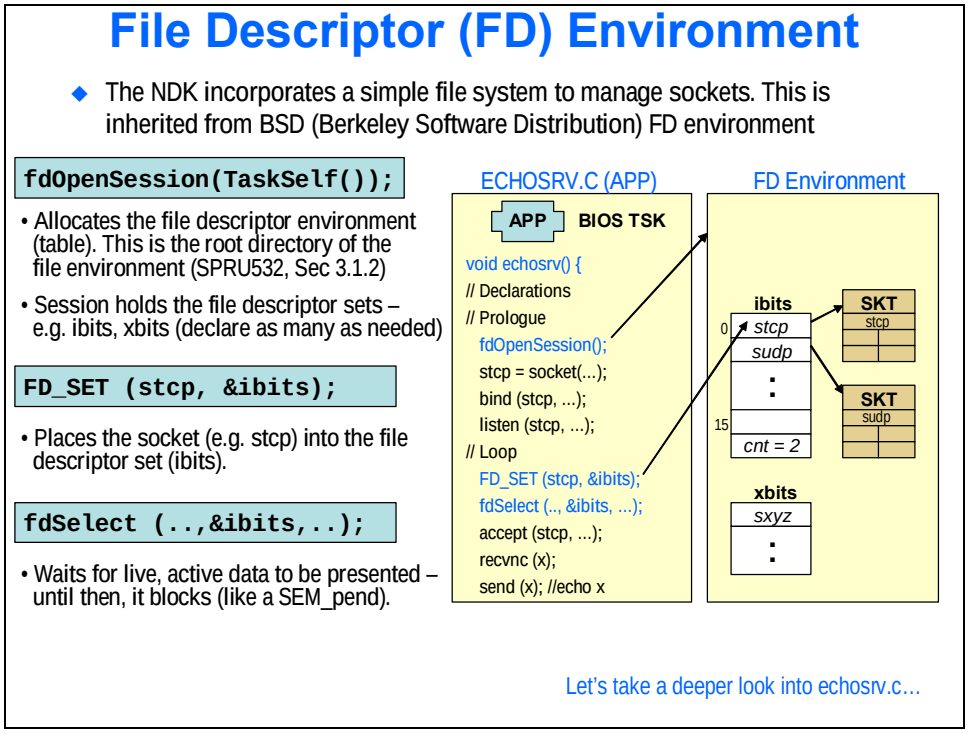
Berkeley Software Distribution (BSD) is the Unix derivative distributed by the University of California, Berkeley. The file descriptor environment is simply a directory structure to keep track of open sockets and is a legacy standard from BSD. The table is created with fdOpenSession() and the “sets” or “directories” in this table are, for example, ibits and xbits. The “sets” point to active sockets (e.g. stcp and sudp).

fdOpenSession() allocates the file descriptor environment (table) (this is the root directory of the file environment). See sec 3.1.2 of Programmer’s guide. The session holds the file descriptor sets – only one session per task – and you can have as many file descriptor sets per session as you desire.

fd_set (typedef) allocates space in the file descriptor environment to hold handles (pointers) to the sockets. ibits is the name of the structure and it defaults to hold 17 items (16 pointers to sockets + a count value of how many sockets are “registered”).

FD_ZERO clears the ibits structure. The FD_SET macro places the first socket (in this case, stcp) into the file descriptor set.(stcp, &ibits). fdSelect() waits for live, active data to be presented – until then, it blocks (just like SEM_pend). This is the BLOCKING part of the TSK.

The file descriptor environment also allows the application to report any errors through fdError(). The table that contains the error codes is located in a header file <serrno.h> located in the NDK include directory.



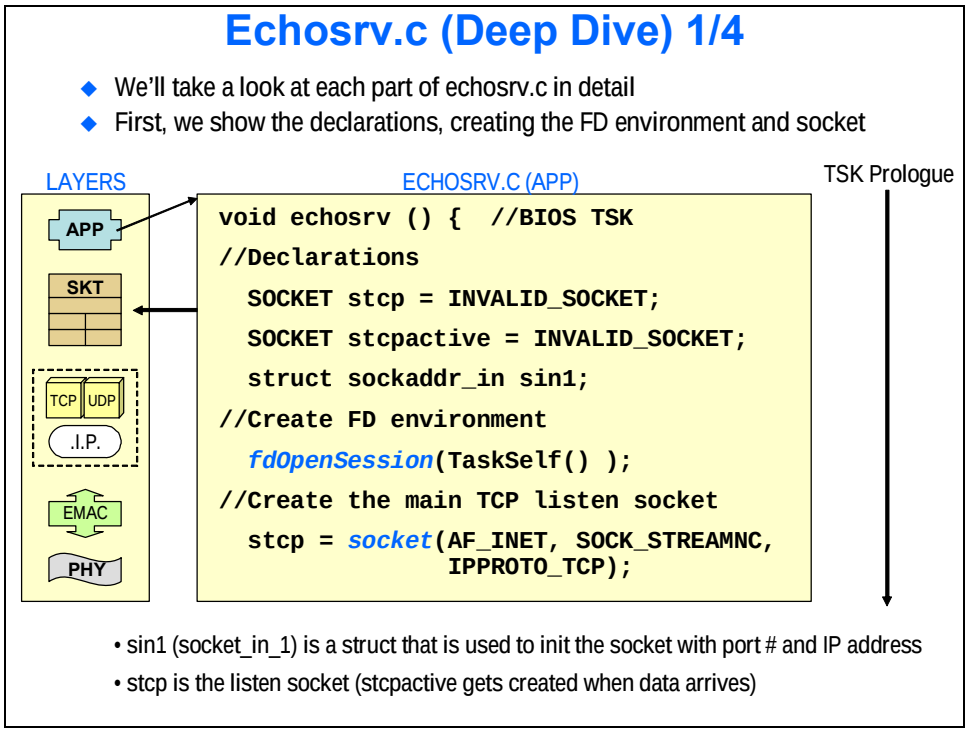
12.5 Echosrv.c – Deep Dive (1/4)

As you go through this discussion on the source file echosrv.c, it would probably be helpful to open the file and view it while reading the following descriptions.

Sockets are declared as INVALID to begin with – this is standard procedure. Sin1 (socket in 1) is a struct that is used to program the socket with the port # and IP address (on the next slide). We already covered what fdOpenSession() does. The sockets are global handles that can be used between child processes (other tasks that are spawned from the task that declared the socket). This is not obvious because of the C language structure (local variables are usually not passed to other functions without the word extern).

Relating back to the TCP flowchart, the first thing that occurs is the socket() API. AF_INET is (address family Internet). SOCK_STREAMNC is the type of socket (streaming or datagram – TCP uses streaming) and the IP protocol is TCP (vs. UDP, which uses datagram).

BIOS TSKs have 3 sections: prologue, loop and epilogue. The prologue runs ONCE (until it hits the blocking function) when the TSK is dynamically created. The loop will then run and unblock based upon the TSK's priority and if the semaphore has been posted.



12.6 Echosrv.c – Deep Dive (2/4)

Bzero() zeroes out the sin1 structure. Sin1 is a structure that is used to write values to ANY socket. Then, we set address family, size and port #. Port 7 is an echo port. Port 21 is FTP. Port 80 is HTTP, etc.

sin1 is the name of the structure – defines variable name sin1 of type struct sockaddr_in.

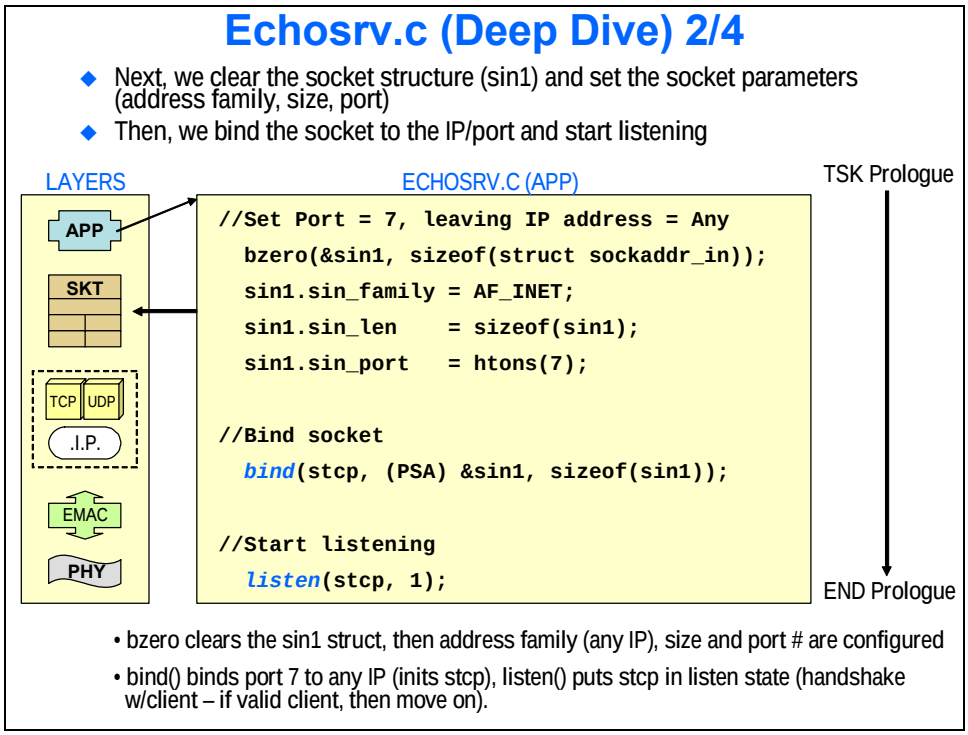
AF_INET name was inherited from BSD as a type to describe the IP address. This is the type of addresses that are arriving on this socket (an IP type of address – if this arrives, it starts listening). We're creating a server, so we want to listen to ANY type of IP address on this socket.

If you want to listen only to a specific IP, you would write:

sin1.in_addr = inet_addr("146.157.4.2");

Bind() takes the structure (sin1) and writes it into the socket (stcp). This is where some of the socket parameters are set (from the previous slide). This effectively binds the port (7) and IP (any) to the socket (stcp). Now, we're ready to listen. PSA is a cast of &sin1 – inherited from the Berkley implementation.

Listen() specifies the socket (stcp). Sets stcp to the "listen" state. This is the end of the prologue of a TCP TSK. ("1" specifies the max number of connections). Listen is the first part of the handshake (it is a valid client, so move on). stcp is always listening. Stcpactive (later on) will be the "active" or "busy" socket.



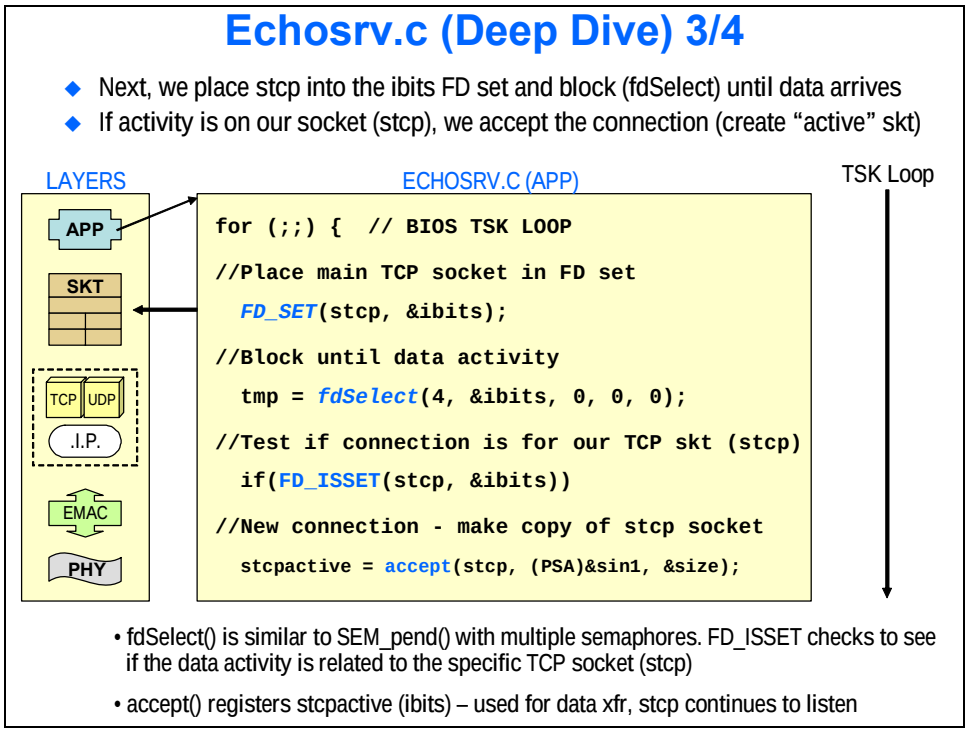
12.7 Echosrv.c – Deep Dive (3/4)

Prior to FD_SET, you need to define ibits as type fd_set and zero ibits using FD_ZERO. FD_SET places the socket (stcp) into the file descriptor set (ibits). This, again, is shown in the diagram on the previous slide.

fdSelect() waits for initial data activity from the client (after the initial handshaking occurred in listen()). The first value “4” is ignored. The 2nd value is for INCOMING ACTIVITY. The 3rd is for outgoing activity. The 4th is for exceptions (out of band – OOB) and the last value is for timeout. fdSelect is really a SEM_pend() with multiple semaphores (any activity on a socket in the file descriptor set – ibits – for incoming, outgoing, etc.).

If (FD_ISSET(stcp, &ibits)) checks to see if there is activity on the specific socket we want – stcp. The previous macro (fdSelect) could have been unblocked for any reason. We want to make sure that it was unblocked for the specific reason of “activity on socket stcp”. If there is activity on stcp, we make a copy of the current socket (stcp) so that stcp can be used elsewhere and stcpactive (or you could call it stcpbusy) can continue processing data and send/receiving data from the client. This is actually analogous to a ping/pong buffer scheme – i.e. one socket is listening and the other is processing – for example, one buffer is being consumed while the other is being written.

Accept() actually registers the new socket into the file descriptor set as well. Now, we have two sockets in ibits (stcp, stcpactive). This is like having a “ping” “ping” buffer – they are the same – one getting data and the other is still listening for other connections.



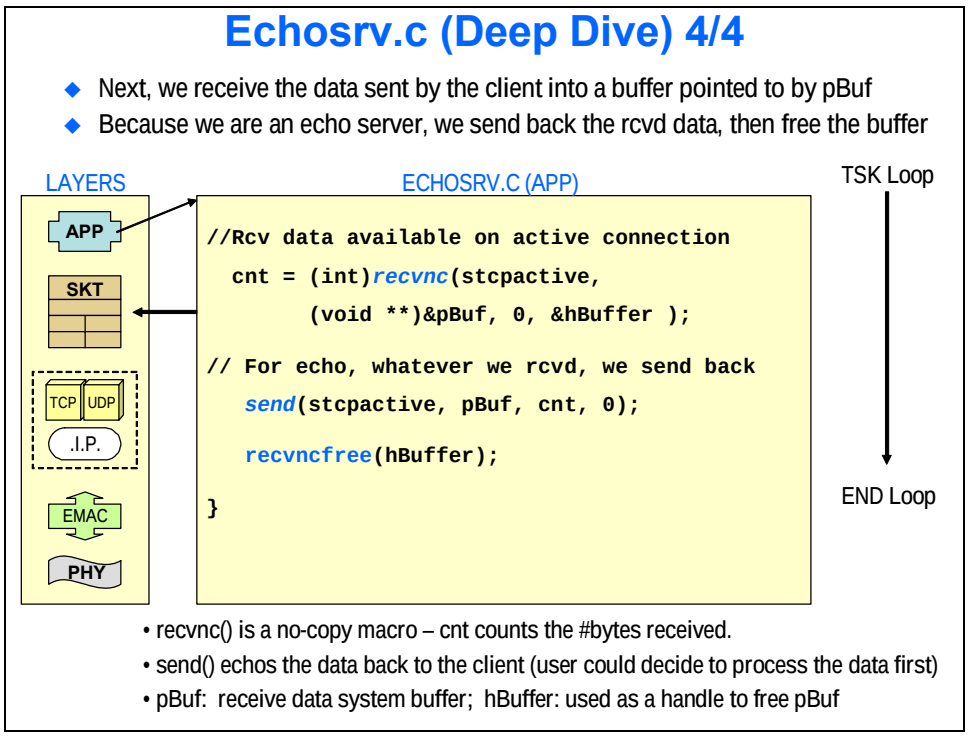
12.8 Echosrv.c – Deep Dive (4/4)

Recvnc (no copy) is a zero-copy macro. Cnt counts the bytes that are received. Recvnc has the following parameters: active socket, pointer to receive data system buffer, flags, handle used to free the pBuf.

Send() sends the data back to the client that sent us the data. This is an echo server. The user could decide to process the data first before sending something back.

Recvncfree (hBuffer) frees the buffer allocated by recvnc back to the system. There are a limited number of buffers that can be stored in the system heap, so this is a must.

This ends the deep dive into sockets programming. Again, for those users that are familiar with sockets programming, the only possible addition is the creation and management of the file descriptor sets. Everything else should seem familiar.



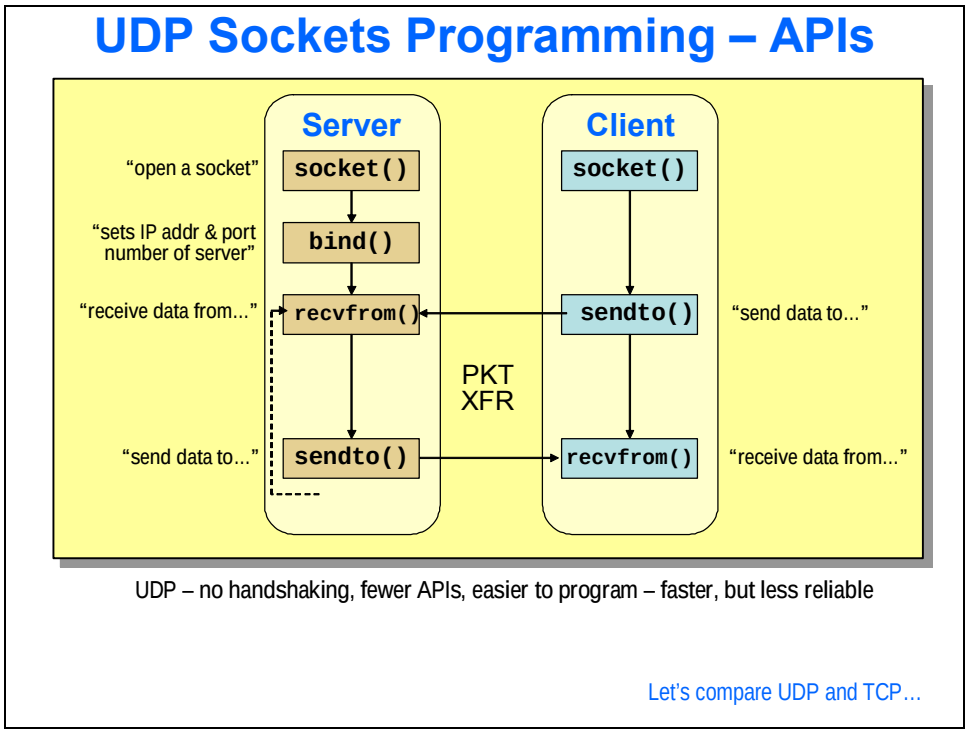
12.9 UDP – Sockets Programming APIs

Compared to the TCP flowchart, notice how simple this diagram is. For the client, it is simply socket(), send/recv. Done. UDP is not a “connected” protocol and therefore does not require the handshaking that TCP does. UDP is therefore faster, but less reliable than TCP. The application developer will pick the desired protocol based on the application’s needs.

For UDP, the server opens a socket, then binds the IP address and port number of the server. Then, it listens for something to arrive.

The client also opens a socket and connects to the server. The client and server then exchange packets.

Notice the absence (vs. TCP) of the following APIs: connect(), listen(), accept().



12.10 TCP vs. UDP

This is a classic speed vs. reliability tradeoff. TCP is a “connected” protocol – there is handshaking involved and delivery is guaranteed. Hence, TCP is slower, but more reliable. If you’re a financial institution (such as an online bank), you will most likely use TCP (if they didn’t use TCP, I think I’d steer away).

UDP has no handshaking and therefore is less reliable, but faster. For example, a streaming site like YouTube would use UDP – fast streaming and if a frame or two drops, who really cares?

So, it is up to the system programmer to decide which protocol is used. Both are easily supported within the NDK.

Sockets Programming – TCP vs. UDP

- **In general, these two protocols do the same thing – send data.**
- **However, their implementation, reliability, speed and usefulness in specific applications vary widely.**
- **Let’s compare each protocol from a high level…**

	TCP	UDP
Connection	• Connected communication protocol • Uses handshaking between hosts • Guarantees reliability of delivery	• No such mechanism • No handshaking • Less reliable (fire and forget)
Speed	Slower	Faster
Socket APIs	Connected: uses connect(), accept(), listen() APIs	Does not use connect(), accept() listen() APIs
Send/Rcv	Uses send() and recv() APIs to the bounded address/port combo	Uses sendto() and recvfrom() APIs where src/dst addr must be specified
Applications	Used in apps that require absolute reliability (financial info, etc.)	Used in apps that don’t care about reliability and speed is critical (e.g. streaming web audio/video)

Let’s compare sockets programming vs. a simple DAEMON server…

12.11 Sockets Programming vs. DAEMON

The key point here is that DAEMONs are easy to use – easy to set up and shut down. They are servers only and therefore cannot initiate activity (like sockets can). DAEMONs are TSKs that are abstracted by the NDK (i.e. there is a TSK living beneath the surface, but they are created and maintained by the NDK).

Sockets programming requires more manual steps than DAEMONs, but the user has full control over exactly how the application works. If you just need a simple server and don't want the hassle of sockets, use a DAEMON. If you want more sophistication and control, then sockets may be the choice.

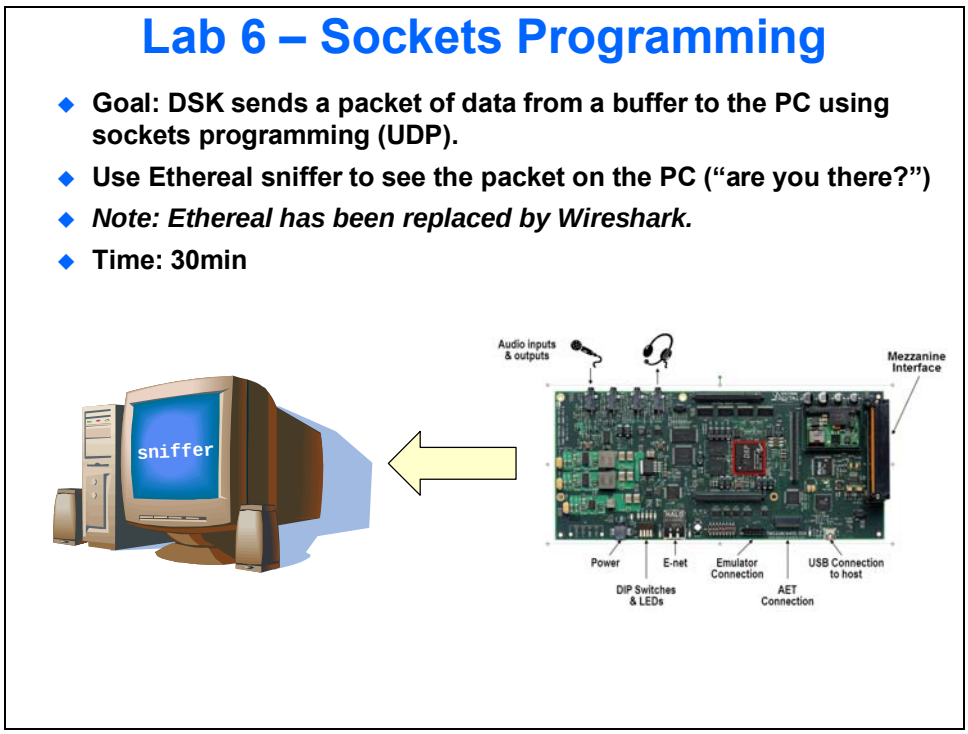
So, basically, DAEMON is a background service that is spawned when data arrives at the proper IP/port. It is a TSK underneath but is completely controlled by the NDK stack. Again, the application developer will make the proper choice based on system needs.

Sockets Programming vs. DAEMON

- In the next lab, we will strip down client.pjt to perform a DAEMON echo server only.
- DAEMONs are TSKs abstracted by the NDK and are dynamically activated only when data arrives (they cannot initiate transfers)
- DAEMON – main advantage is reuse of memory – also coding is more efficient
- Sockets Programming uses DSP/BIOS TSKs and a file descriptor environment (they run forever) and can initiate a transfer (unlike a DAEMON)

	Sockets Programming	DAEMON
Type	• Created as BIOS TSKs • TSKCreate() • Runs continuously and requires fdSelect() to wait on received data	• Creation requires port, IP, UDP or TCP • DaemonNew() • Created dynamically when data arrives on a specific port/IP combo
File Descriptor	• Requires a file descriptor • fdOpenSession()	• No file descriptor needed
Sync	• Requires sync mechanism • FD_SET, FD_CLR, fdSelect()	• Mechanism is embedded in the stack
Sockets	• Used/declared in function body • socket(), bind(), connect(), etc.	• Passed as parameter • Typically used for data recv/send

13 Lab 6 – Use Sockets Programming APIs



13.1 Lab Overview:

- In the last lab, we took the easy route – a DAEMON echo server. However, DAEMONs cannot initiate transfers – they can only respond to clients and send data after establishing a connection. On the other hand, using sockets programming APIs, we can build a client application that can initiate a transfer.
- That's the goal of this lab: to use the NDK to create the DSK as the client and send a message to the PC (server). We will start with the code we used in the last lab and build on it. To minimize typing mistakes, we have two starter files: sender.c and addToClient_starter.c (both located in the \starter directory). Sender.c contains the sockets programming APIs to send a packet and lab8C_starter.c contains a few commands we need to copy and add to client.c. Although you are not writing this application from scratch, it will give you a good feel for how to work with the NDK to do whatever your application requires.

13.2 Use Sockets Programming to SEND Data

1. Open CCS and open client.pjt.

Just a reminder here. We are currently modifying the original NDK code. If, for some reason, we wanted to go back to the first lab and run it, we couldn't. But remember, we saved a copy of the "example" directory so that we can retrieve any files that might get corrupted or broken – smart idea. If you purchase the NDK, remember to do this right away before modifying files. It will save a few Ibuprofen.

Open CCS and open client.pjt.

We only have a few modifications to turn the DSK into a client and send a message to the PC. Most of the work has been done for you in the starter files. And, based on the discussion material, you should understand most of what sender.c is doing. Also, keep in mind that the code we are using sets up a UDP client – we did the TCP code in the discussion material. Ok, let's go make this thing work…

2. Add sender.c to your project.

Add sender.c (located in the \starter directory) to your project. Open sender.c and browse its contents. What you will see should look familiar based on the discussion material. The task that gets created and used is udp_test. Notice the familiar sockets programming APIs inside the task.

Locate the buffer: sendBuffer. Write down below what it contains:

sendBuffer = _____

3. Delete unused code in client.c

We are starting with the client.c from our last lab (DAEMON echo), so there is code that will go unused in this lab. Let's clean that up first:

Near the top of client.c, remove the following two lines of code:

```
#include <common/console/console.h>
#include <common/servers/servers.h>
```

In NetworkOpen(), remove the two calls to DaemonNew(). So, in essence, you have an empty function here. We'll add a task create to NetworkOpen() shortly.

In NetworkClose(), remove the two calls to DaemonFree(). Again, we'll add a line of code to this function shortly.

Above NetworkOpen(), remove the declaration of these handles:

```
static HANDLE hEcho=0,hEchoUdp=0;
```

4. **Modify client.c to include init code for the task in sender.c.**

   Every 5 seconds, sender.c sends a packet with the contents of sendBuffer to the PC. We only need to add a few items to client.c to get this to work.

   Open the file addToClient_starter.c from the \starter directory. Also, open client.c for editing. Position the windows so that you have both open at the same time.

   - First, we have to create the task (udp_test) that is used in sender.c. We'll use the API TaskCreate() to do this. Copy the first line (LINE 1) into the NetworkOpen() function in client.c (near line 240 or so).
   - Next, for each task created, we need to make sure to destroy it in the event of a stack shutdown. Insert LINE2 (TaskDestroy) into the NetworkClose() function in client.c (near line 250).
   - Near the top of client.c, declare the global handle that is returned by TaskCreate(). Add LINE3 (the declaration for hUDPServer), just beneath this declaration:

     ```
     static void ServiceReport(uint Item, uint Status, uint Report, HANDLE hCfgEntry );
     ```

   - The last step is to add the prototype for the C function that corresponds to the task. Add LINE4 near the top of client.c to the "external references" area of the code. Close addToClient_starter.c. We're now ready to compile and run.

5. **Rebuild All and run.**

   Rebuild client.pjt and fix any errors that occur. When you get a clean build and the program loads, run it. In the command window in CCS, you should see the message "message sent to Ethernet". If you don't see this, ask your instructor for help. If you do see this, congrats – you're on the right track. The DSK is now continuously (every 5 seconds) sending the message "Are you there?" to the PC – that is the msg inside sendBuffer.

6. **Use Wireshark to see the data coming from the DSK.**

   Open Wireshark and capture the network traffic. You should see the UDP packet count increase every 5 seconds while capturing. After about 20 seconds, hit the Stop button. Click on one of the echo packets. You should see UDP packets from 192.168.1.41 that contain the string "Are you there?" If not, ask your instructor for help (or, debug the problem yourself – your instructor probably has better things to do). ☺

13.3 Save Your Solution and Close CCS

7. Save the following files using Windows Explorer:

- \example\network\client\dsk6455\bin\client.out (name it: client-lab6.out)
- \example\network\client\common\client.c (name it: client-lab6.c)

Into the following directory:

```
C:\IW64x+\solutions_student\Lab6
```

You may need to create this directory using Windows Explorer and then copy your files.

Close CCS.

You're Done

14 Conclusion

14.1 NDK Considerations

There is a TON of useful information in the NDK User Guide (SPRU523). Many MANY questions from users are answered there. This is a list of the top 9 issues users have that they stumble on. Instead of just copying text out of the user guide, pointers to answers are provided here.

After reading this getting started guide, the next step for the user should be to investigate each of the 9 issues highlighted below. After that, it really depends on the user's specific needs. We cannot emphasize enough how important it is to read these specific sections in the NDK User Guide. It will save you many headaches and valuable time.

The new app note (SPRAAQ5) discusses all of the benchmark information for the NDK – how long each function takes, memory usage, optimization tricks, etc.

NDK Ver 1.92 Considerations

- Most of the following is contained in the NDK User Guide (SPRU523). However, it is a useful list of "pointers" to information that may help your initial out-of-box experience with the NDK:
  - Library descriptions (SPRU523, Sec 1.3)
  - Explanation of the NDK software directory structure (SPRU523, Sec 1.3.3)
  - NDK Initialization and Configuration (SPRU523, Sec 3.3)
  - HAL drivers assume some L2 cache is configured (SPRAAQ5)
  - Stack size recommendations (UDP/TCP stack sizes, SPRU523, Sec 3.2.2.1)
  - Choosing TSK priorities (SPRU523, Sec 3.2.2.2)
  - I am getting "this" error – what do I do now? (SPRU523, Sec 3.4)
  - Packet buffers are allocated in a memory section called NDK_PACKETMEM configurable by the user (SPRU523, Sec 3.1.4)
  - Use client.pjt as your starting point for your application (many "uh oh's" can be avoided following this simple advice vs. starting with a blank page)

New NDK version 1.94…

14.2 Update – New NDK version 1.94

Update – NDK Ver 1.94 Now Available

- As this document was being authored and reviewed, two newer versions of the NDK were released: 1.93 and 1.94. Following is a list of the improvements in Ver. 1.94:
  - Bug fixes in NDK 1.93 release
  - Support for multiple interfaces *(enables you to simultaneously attach multiple device drivers to the NDK stack – e.g. a native EMAC driver + external WLAN driver)*
  - Limited IEEE 802.1p packet priority marking *(useful if the "receiver" of packets from the NDK needs to do any type of processing based on packet markings)*
  - Nettools Library shipped in binary AND source code format *(this allows you to recompile the nettools library after removing unwanted services – e.g. DHCP, server/client, TFTP, HTTP, PPP, etc. – smaller code size)*
  - Validated on the following platforms (only BIOS 5.3): DM642 (C64x), C6455 (C64x+)
  - Note: DM6437 – Ethernet drivers have not been modified for this device. So, the new features are NOT available at this time.

For more information…

14.3 For More Information…

For More Information…

- Shown below are references that you can use to learn more about networking, the NDK and the EMAC:

• User Guides and Application Notes:

NDK User Guide – SPRU523

NDK Programmer's Guide – SPRU524 (all the API calls are listed here)

EMAC User Guide – SPRU975

Using IP Multicasting with the TMS320C6000 NDK – SPRAAI3

NDK Benchmarks – SPRAAQ5

Getting Started with the C6000 Network Development Kit (NDK) – SPRAAX4

• Useful Textbooks:

UNIX Network Programming – vol 1 (The sockets and networking API). Stevens, Richard W.; Fenner, Bill; Rudoff, Andrew M. Third Edition.

TCP/IP Illustrated – vol 1 (The protocols). Stevens, Richard W.

TCP/IP Protocol Suite. Forouzan, Behrouz A.

• Internetworking Technology Handbook (CISCO Systems)

`www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/index.htm`

TI NDK Support: *http://tiexpressdsp.com/wiki/index.php?title=Category:NDK*

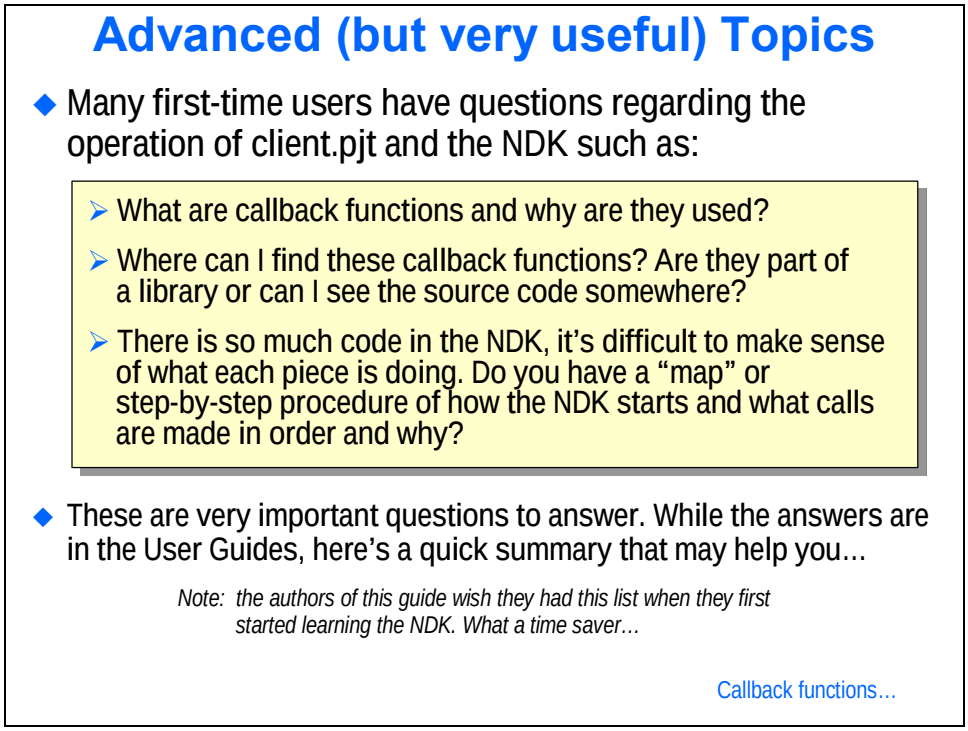
Advanced (but useful) topics…

15 Advanced (but useful) Topics

15.1 Introduction

Following is some additional information added late in the authoring process. Users ask many questions regarding these topics to our support line and forums. Therefore, we just couldn't resist adding this hopefully helpful information. "Advanced" may not be the right word – so, if you're new to the NDK, this might be exactly what you needed. If you're a networking guru (advanced user), it could still be exactly what you needed. ☺

Regardless, it's worth knowing this info is here and may help avoid users like you jamming up our support lines with questions… ☺



15.2 Callback Functions

Callback Functions

- A callback function is a function that is called *from* the device driver (e.g. the EMAC driver located in a .lib file) to source code that helps verify that it is working correctly.
- In client.pjt, locate the source file "dsk6455init.c". This file contains two callback functions (that are called by the EMAC driver):
  - **`C6455EMAC_getConfig()`**

    *This function reports that the MAC address was properly assigned to the EMAC and allows you to configure which CPU interrupt the EMAC peripheral uses – look for the variable *pIntVector.*
  - **`C6455EMAC_linkStatus()`**

    This function is used to report that a valid link was detected and at what speed.

"Map" of how the NDK starts up and what happens next…

15.3 Order of Events (inside the NDK…)

Order of Events (inside the NDK…)

- Following is a list of events in the order they occur. This list is not complete, but highlights the areas of main concern regarding the NDK. For hardware reset events, refer to your device's data sheet.

1. Hardware Reset, Boot Sequence
2. Reset vector points to **`_c_int00`** which is the *BIOS_init()* function (if BIOS is enabled)
3. *C6455_init()* in file **`dsk6455init.c`** is called by BIOS initialization code (configured in the .tcf file under System ? General Settings)
4. *main()* is called. *main()* runs and then either return()s from *main()* or falls out of *main()* (without this, DSP/BIOS and the scheduler will never run).
5. *BIOS_start()* runs and finishes its initialization. Then BIOS (scheduler) starts running.

Continued…

Order of Events (inside the NDK…)

- Following is a list of events in the order they occur. This list is not complete, but highlights the areas of main concern regarding the NDK. For hardware reset events, refer to your device's data sheet.

6. *StackTest()* is called (it is a static TSK configured in the **`.tcf`** file). A msg is printed to stdout: "TCP/IP Stack Example Client"
7. *NC_NetStart()* is called by *StackTest()* and initializes the stack (both device drivers, NDK scheduler and TCP/IP)
8. *C6455EMAC_getConfig()* is called and a msg is sent to stdout: "Using MAC Address: 00-01-02-03-04-05" (or whatever number is configured)
9. *ServiceReport()* in **`client.c`** is called as many times as the number of services that need configuration in StackTest(). Messages are printed to stdout following this format:

```
Service Status:  DHCPC    : Enabled :       : 000
Service Status:  Telnet   : Enabled :       : 000
```

Continued…

Order of Events (inside the NDK…)

- Following is a list of events in the order they occur. This list is not complete, but highlights the areas of main concern regarding the NDK. For hardware reset events, refer to your device's data sheet.

10. *NetworkOpen()* is called and initializes dynamically all the TSKs and DAEMONs needed by your application. By default, in client.pjt, you will see five DAEMONs configured
11. *ServiceReport()* is called once again (if DHCP is enabled) to report that this service is properly configured and running. A msg is printed to stdout (e.g.):

```
Service Status: DHCPC   : Enabled   : Running   : 000
```

12. *C6455EMAC_IinkStatus()* is then called by the device driver after a proper link is detected by the PHY. A msg is sent to stdout (e.g.):

```
Link Status: 100Mb/s Full Duplex on PHY 1
```

Continued…

Order of Events (inside the NDK…)

- Following is a list of events in the order they occur. This list is not complete, but highlights the areas of main concern regarding the NDK. For hardware reset events, refer to your device's data sheet.

13. *NetworkIPAddr()* is then called when an IP address is assigned by the DHCP server (or is statically configured in **`client.c`**). A msg is sent to stdout (e.g.):

```
Network Added:  If-1:192.168.0.2
```

14. The network stack then starts running.

15. Additional details can be found in section 3.3 of the User Guide.

Ok…enough details…you're on your own now… ☺

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Clocks and Timers	www.ti.com/clocks	Digital Control	www.ti.com/digitalcontrol
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
RFID	www.ti-rfid.com	Telephony	www.ti.com/telephony
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265