TI Network Developer's Kit (NDK) v2.21

User's Guide

Literature Number: SPRU523H
May 2001–Revised February 2012

Contents

List of Figures

List of Tables

Read This First

About This Manual

The document covers NDK programming as it applies to the TMS320C6000, Cortex-A8, and ARM9 programming environment, including Code Composer Studio™ (CCStudio) Development Tools. It is not intended as an API reference. This manual also provides necessary information regarding how to effectively install, build, and use the Network Developer's Kit (NDK) in user systems and applications.

The latest version number as of the publication of this guide is NDK v2.21.

How to Use This Manual

The information presented in this document is divided into the following chapters:

- **Chapter 1: Overview** introduces the stack and developing network applications.
- **Chapter 2: Example Applications** provides examples that are good for platform test and demonstration, and also serve as a good starting point for developing your own network applications.
- **Chapter 3: Network Application Development** describes the NDK software, and how to start developing network applications now.
- **Chapter 4: Network Control Functions** describes the internal workings of the network control layer (NETCTRL).
- **Chapter 5: OS Adaptation Layer: OS and MiniPrintf** describes the OS adaptation layer that controls how the NDK uses SYS/BIOS resources. This includes Tasks, Semaphores, memory and printing. Anything that is related to OS can be adjusted here. While the NDK is built for the SYS/BIOS operating system, the NDK OS Adaption Layer can be modified to support other operating systems, if you so desire.
- **Appendix A: Revision History** describes the changes to this document since the previous release.

Notational Conventions

This document uses the following conventions:

- Program listings, program examples, and interactive displays are shown in a special typeface.
- In syntax descriptions, the function or macro appears in a bold typeface and the parameters appear in plainface within parentheses. Portions of a syntax that are in bold should be entered as shown; portions of a syntax that are within parentheses describe the type of information that should be entered.
- Macro names are written in uppercase text; function names are written in lowercase.

Related Documentation From Texas Instruments

Additional information about the NDK can be found in SPRU524 (*TI Network Developer's Kit (NDK) API Reference Guide.*) and the NDK category of the TI Embedded Processors Wiki. If you have questions, you can ask them on the BIOS forum in TI's E2E community.

Information about SYS/BIOS, which is used in NDK applications, can be found in the SPRUEX3 (*TI SYS/BIOS Real-time Operating System User's Guide*) and the SYS/BIOS main page of the TI Embedded Processors Wiki.

The following documents describe Cortex™-A8 and ARM9 devices and related support tools. Many of these documents can be found on the Internet at http://www.ti.com.

SPNU151— *ARM Optimizing C/C++ Compiler User's Guide*

SPNU118— *ARM Assembly Language Tools User's Guide*

SPRUH73— *AM335x ARM® Cortex™-A8 Microprocessors (MPUs) Technical Reference Manual*

Cortex-A8 wiki page—on the TI Embedded Processors Wiki

ARM9 wiki page—on Ti's Embedded Processors Wiki

Sitara ARM Microprocessors forum—in TI's E2E Community

The following documents describe the TMS320C6x™ devices and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477–8924. When ordering, please identify the book by its title and literature number. Many of these documents can be found on the Internet at http://www.ti.com.

SPRU189 — *TMS320C6000 DSP CPU and Instruction Set Reference Guide.*

SPRU190 — *TMS320C6000 DSP Peripherals Overview Reference Guide.*

SPRU197 — *TMS320C6000 Technical Brief.*

SPRU198 — *TMS320C6000 Programmer's Guide*

SPRU509 — *TMS320C6000 Code Composer Studio ™Development Tools v3.3 Getting Started Guide*

SPRUFP2 — *TMS320C6000 Network Developer's Kit (NDK) Support Package Ethernet Driver Design Guide.*

Overview

This chapter introduces the Network Developer's Kit (NDK) by providing a brief overview of the purpose and construction of the NDK, along with hardware and software environment specifics in the context of NDK deployment. This Network Developer's Kit (NDK) Software User's Guide serves as an introduction to both the NDK and to developing network applications.

Topic **Page**

1.1 Introduction

The Network Developer's Kit (NDK) is a platform for development and demonstration of network enabled applications on TI embedded processors, currently limited to the TMS320C6000 DSP family and ARM processors. The code included in this NDK release is generic C code which runs on any C64P, C66, C674, ARM9, Cortex-A8, or Cortex-M3 device (for C6000 processors, both big-endian and little-endian modes are supported). The NDK stack serves as a rapid prototyping platform for the development of network and packet processing applications. It can be used to add network connectivity to existing applications for communications, configuration, and control. Using the components provided in the NDK, developers can quickly move from development concepts to working implementations attached to the network.

The NDK is a networking stack that operates on top of the SYS/BIOS Real-Time Operating System (RTOS). The stack can be ported to any hardware in the TMS320C6000, Cortex-A8, and ARM9 families. The NDK software package is designed to be a transparent add-on to SYS/BIOS and Code Composer Studio ™Development Tools.

1.2 Installing and Setting Up the NDK

The NDK is designed to be used with Code Composer Studio (CCS). While it is compatible with CCS v4.2, it is recommended that you install CCS v5.1 or higher for use with the NDK. The instructions and figures in this document are for use with CCS v5.1. It is best to install the version of CCS that you will use with the NDK before installing the NDK.

The NDK product is provided as a zipped archive. You can download it from TI's Embedded Software Download Page.

Unzip the downloaded file (on Windows or Linux) in the directory where you installed Code Composer Studio (CCS). For example, if you unzip the downloaded NDK file in the c:\ti directory on Windows, the <NDK_INSTALL_DIR> will be c:\ti\ndk_#_##_##_##, where # is a digit in the version number.

In addition to downloading the NDK, you will need to download a NDK Support Package (NSP). The NSP contains Ethernet driver code, libraries and network examples for various TI platforms. It is meant to supplement the Network Development Kit (NDK), which consists of the platform-independent networking code. The Embedded Software Download Page provides links to download the NSP. The Embedded Software Download Page provides a link to download the NSP for the OMAP-L138. Other NSPs are provided as part of the SDK for that development platform.

Unzip the NSP file in same the directory where you installed Code Composer Studio (CCS). For example, if you unzip the downloaded NSP file in the c:\ti directory on Windows, the <NSP_INSTALL_DIR> will be c:\ti\nsp_#_##_##_##, where # is a digit in the version number.

After you unzip both the NDK and the NSP, start CCS. You will be notified that CCS found these two products and asked if you want to use them. You need to restart CCS in order to make new products available within CCS.

Network application development is typically performed in Code Composer Studio (CCStudio). The stack libraries are designed to work with Code Composer Studio. If you have experience creating makefiles, you can write and use your own makefiles to build a network application. However, the NDK does not provide example makefiles for building from the command line.

1.2.1 Rebuilding NDK Libraries

The NDK installation includes all source files and full support for rebuilding its libraries. In order to rebuild the NDK libraries, please see the instructions in the Rebuilding the NDK Core topic in the TI Embedded Processors Wiki.

1.3 NDK Stack Library Design

The NDK was designed to provide a full TCP/IP functional environment, with or without routing, in a small memory footprint.

1.3.1 Design Philosophy

The NDK is isolated from both the native OS and the low-level hardware by abstracted programming interfaces. The native OS is abstracted by an operating system adaptation layer (OS), and custom hardware is supported via a hardware adaptation layer (HAL) library. These libraries are used to interface the stack to SYS/BIOS and to the system peripherals.

The NDK stack was originally designed to be able to communicate with a single device driver at a time. This was called the *LL Packet Driver* architecture. The Network Interface Management Unit (NIMU) architecture has been introduced starting from the NDK 1.94 release to overcome this limitation of the LL packet driver architecture. NIMU provides an interface between the stack and the device drivers through which the stack can talk to multiple instances of a single or various device drivers concurrently.

In comparison to single-port serial device applications, the new NIMU architecture is best suited for Ethernet type devices where it's most common to have multiple instances running concurrently. LL Packet driver architecture is no longer supported from NDK v2.0 release onwards. All the core NDK libraries and the supporting NSP Ethernet driver libraries for the platforms are NIMU compliant now. To obtain LL compliant drivers, one would have to obtain them from an older NDK release.

Various features like virtual LAN (VLAN), Raw Ethernet socket and IPv6 stack support are controlled by NIMU. These features are available only with NIMU-enabled application and libraries. VLAN and Raw Ethernet socket support are by default built in to a NIMU-enabled stack library. VLAN support enables the stack to receive, process, and transmit VLAN tagged packets. Similarly, support for Raw Ethernet sockets (different from Raw IPv4/IPv6 sockets) enables any application using the NDK stack to send/receive Ethernet packets with custom Layer 2 (L2) protocol type, i.e., protocol type in the Ethernet header of the packet other than any of the well known standard protocol types like IP (0x800), IPv6 (0x806), VLAN (0x8100), PPPoE Control (0x8863) or PPPoE Data (0x8864). The stack is available for both IPv6 and IPv4.

The NDK core stack can support Jumbo frames (packet sizes larger than 1500 bytes) also. The Jumbo frame support can be built into an application by linking with libraries compiled for Jumbo frame support. The libraries and application would have to be recompiled with the following pre-processor definition added: _INCLUDE_JUMBOFRAME_SUPPORT.

For more details on VLAN, IPv6, Raw Ethernet sockets and Jumbo frames support, see the *TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524).

1.3.2 Control Flow

Figure 1 shows a conceptual diagram of how the stack package is organized in terms of function call control flow. The five main libraries that make up the NDK are shown. These are STACK, NETTOOL, OS and MiniPrintf, HAL, and NETCTRL. These libraries are summarized in sections that follow. NIMU related changes are also discussed in the affected libraries (STACK, NETCTRL, and NETTOOL).

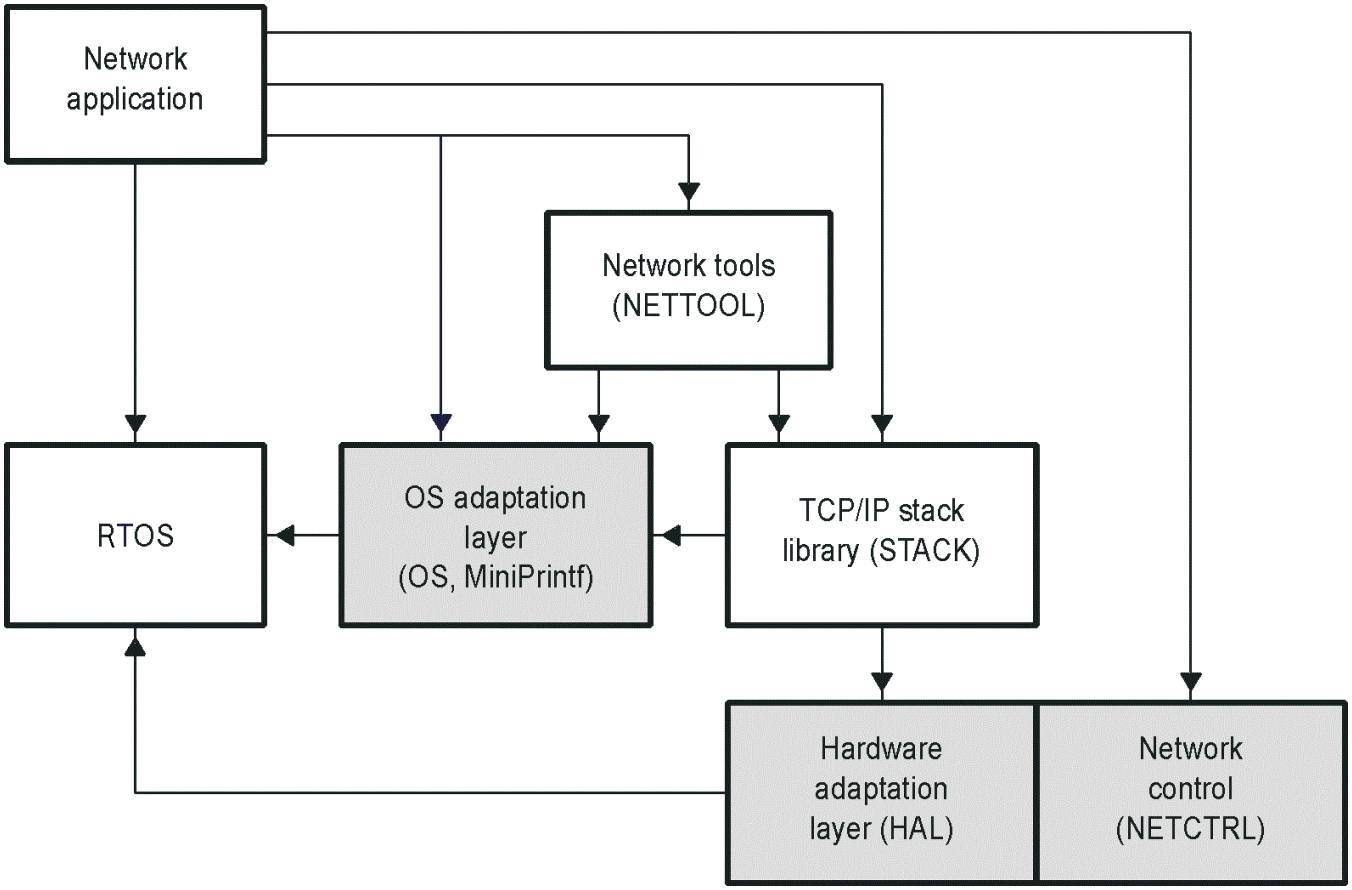


Figure 1-1. Stack Control Flow

1.3.3 Library Directory Structure

Pre-built linkable libraries and source code are provided for each of the libraries that make up the NDK in the <NDK_INSTALL_DIR>\packages\ti\ndk directory tree. The pre-built libraries are in a lib subdirectory of the directory for each library. See Section 1.5.1 for a list of the directories in <NDK_INSTALL_DIR>\packages\ti\ndk.

- Both IPv4 and IPv6 libraries are provided. Filenames that do not include "ipv4" are compiled for IPv6. However, in the <NDK_INSTALL_DIR>\packages\ti\ndk\stack\lib directory, filenames that do not include "6" are compiled for IPv4.
- The NETCTRL library comes in "min", regular, and "full" versions. For example, netctrl_min, netctrl, and netctrl_full. See Section 1.3.8 for details.
- For platforms where it is supported, both little-endian and big-endian libraries are provided. Libraries with an "e" in the file extension are big-endian libraries. All others are little-endian.
- Libraries with Jumbo Frame support (for packet sizes larger than 1500 bytes) are not included in the NDK installation. If you want NDK libraries with Jumbo Frame support enabled, you will need to #define the _INCLUDE_JUMBOFRAME_SUPPORT pre-processor definition and rebuild the libraries as described in the Rebuilding the NDK Core topic in the TI Embedded Processors Wiki.

The file extensions for pre-built libraries provided with the NDK are as follows:

.a64P	For C64x+ targets (COFF format, little endian).
.a64Pe	For C64x+ targets (COFF format, big endian).
.a674	For C674x targets (COFF format, little endian).
.ae9	For ARM9 targets (ELF format, little endian)
.ae66	For C66x targets (ELF format, little endian).
.ae66e	For C66x targets (ELF format, big endian).
.ae674	For C674x targets (ELF format, little endian).
.aea8f	For Cortex-A8 targets (ELF format, little endian, does not use hardware-based floating point support, for legacy application support)
.aea8fnv	For Cortex-A8 targets (ELF format, little endian, uses hardware-based vector floating point support, recommended over .aea8f)
.aem3	For Cortex-M3 targets (ELF format, little endian)

The libraries provided with the NDK are platform independent. That is, versions of these libraries are provided for all platforms. Any hardware-dependent libraries that exist only for certain platforms are distributed in the appropriate NDK Support Package (NSP), which you download separately from the NDK.

The NDK installation includes all source files and full support for rebuilding its libraries. In order to rebuild the NDK libraries, please see the instructions in the Rebuilding the NDK Core topic in the TI Embedded Processors Wiki.

1.3.4 The STACK Library

The STACK library is the main TCP/IP networking stack. It contains everything from the sockets layer at the top to the Ethernet and Point-to-point protocol (PPP) layers at the bottom. The library is compiled to make use of the SYS/BIOS operating system, and does not need to be ported when moved from one platform to another. Several builds of the library are included in the NDK.

The STACK libraries are provided in the <NDK_INSTALL_DIR>\packages\ti\ndk\stack\lib directory. The following versions of the library either include or exclude features like PPP, PPP over Ethernet (PPPoE), and Network Address Translation (NAT).

Library	Variants	Description
STK	stk, stk6	Stack with NIMU, VLAN, and Raw Ethernet Socket support. Variants support IPv4 or IPv6. No support for PPP, PPPoE, NAT and LL architectures.
	stk_nat, stk6_nat	Stack with NAT, NIMU, VLAN, and Raw Ethernet Socket support. Variants support IPv4 or IPv6. No support for PPP, PPPoE, and LL architectures.
	stk_nat_ppp, stk6_nat_ppp	Stack with NAT, PPP, NIMU, VLAN, and Raw Ethernet Socket support. Variants support IPv4 or IPv6. No support for PPPoE and LL architectures.
	stk_nat_ppp_pppoe, stk6_nat_ppp_pppoe	Stack with NAT, PPP, PPPoE, NIMU, VLAN, and Raw Ethernet Socket support. Variants support IPv4 or IPv6. No support for LL architectures.
	stk_ppp, stk6_ppp	Stack with PPP, NIMU, VLAN, and Raw Ethernet Socket support. Variants support IPv4 or IPv6. No support for NAT, PPPoE and LL architectures.
	stk_ppp_pppoe, stk6_ppp_pppoe	Stack with PPP, PPPoE, NIMU, VLAN, and Raw Ethernet Socket support. Variants support IPv4 or IPv6. No support for NAT and LL architectures.

This library has been modified to additionally support NIMU core architecture, VLANs, IPv6, Jumbo Frames and Raw Ethernet sockets. The NIMU core replaces the Ethernet and IF objects which existed previously and provides the following services:

- Generates unique names and identifiers for each NIMU network interface object. The unique name is an extension to the previous LL packet layer architecture which only used device indexes for identification.
- Provides an interface used by the drivers to pass Ethernet packets up to the NDK core stack. The function supports handling of 802.3 and 802.1Q tags.
- Provides a configuration interface which allows the ability to configure receive filters/multicast addresses, etc. This is done through the IOCTL interface API.
- Polls the various NIMU network Interface objects registered with it.
- Ensures that packets are allocated with sufficient head and tail room for customized headers which need to be inserted by various layers.

The NIMU-enabled NDK stack also has support for VLAN processing by default, as mentioned earlier. The NIMU core and VLAN modules in the NDK stack are very closely tied in. The VLAN module is brought down by the NIMU as a part of its de-initializing routine during the system shutdown. Similarly, the NIMU module's packet receive routine is responsible for handing over all VLAN tagged packets to the VLAN module for processing. It does so by checking all incoming packets for their L2 type and if it is VLAN, forwards the packets onto the VLAN module of the stack for further processing.

1.3.5 NETTOOL Libraries

The Network Tools (NETTOOL) function library contains all the sockets-based network services supplied with the NDK, plus a few additional tools designed to aid in the development of network applications. The most frequently used component in the NETTOOL library is the tag-based configuration system. The configuration system controls nearly every facet of the stack and its services. Configurations can be stored in non-volatile RAM for auto-loading at BOOT time.

The NETTOOL libraries are provided in the <NDK_INSTALL_DIR>\packages\ti\ndk\nettools\lib directory.

The tools provided in the NETTOOL library use the NIMU IOCTL calls directly to retrieve device-related information.

See Section 1.4.3 for information about the NETTOOL services and APIs.

1.3.6 OS and MiniPrintf Libraries

These libraries form a thin adaptation layer that maps some abstracted OS function calls to SYS/BIOS function calls. This adaptation layer allows the SYS/BIOS system programmer to tune the NDK system to any OS based on SYS/BIOS. This includes Task thread management, memory allocation, packet buffer management, printing, logging, critical sectioning, cache coherency, interrupt management and jumbo packet buffer management.

The OS libraries are provided in the <NDK_INSTALL_DIR>\packages\ti\ndk\os\lib directory. The "os" library is the OS Adaptation Layer library with priority exclusion. The "os_sem" library uses semaphore exclusion, instead. See Section 1.4.1 for information about the OS Adaptation Layer's use of Task and Semaphore objects.

The MiniPrintf library provides printing functions with a small code size to help keep the application's footprint small. These are packaged into a separate library, so you can use either the full-fledged RTS printing functions provided with the Code Composer Studio or the small functions included with the MiniPrintf library.

The MiniPrintf libraries are provided in the <NDK_INSTALL_DIR>\packages\ti\ndk\miniPrintf\lib directory.

1.3.7 HAL Libraries

The HAL libraries contain files that interface the hardware peripherals to the NDK. These include timers, LED indicators, Ethernet devices, and serial ports. The drivers contained in the <NDK_INSTALL_DIR>\packages\ti\ndk\hal directory are as follows:

Library	Description
eth_stub\lib\hal_eth_stub	Ethernet Stub Driver
ser_stub\lib\hal_ser_stub	Serial Stub Driver
timer_bios\lib\hal_timer_bios	Timer Driver Using SYS/BIOS Timer object
userled_stub\lib\hal_userled_stub	User LED Stub Driver

See Section 1.4.5 for information about the HAL APIs. The HAL is also discussed in the *TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524) and the *NDK Support Package Ethernet Driver Design Guide* (SPRUFP2).

1.3.8 NETCTRL Libraries

The NETCTRL or Network Control library can be considered the center of the stack. It controls the interaction between the TCP/IP and the outside world. Of all the stack modules, it is the most important to the operation of the NDK. Its responsibilities include:

- Initializing the NDK and low-level device drivers
- Booting and maintaining system configuration via configuration service provider callback functions
- Interfacing to the low-level device drivers and scheduling driver events to call into the NDK
- Unloading the system configuration and driver cleanup on exit

The NETCTRL library has been modified for NIMU and related feature support to do the following:

- Initialize the NIMU core during stack bring-up, which in turn initializes and starts all the device drivers registered with the NIMU core. Initializes the VLAN module in the NDK core stack.
- De-initialize NIMU core during stack shutdown, which in turn cycles through all the registered device drivers and shuts them down.
- Polls all the registered devices periodically so as to allow them to perform any routine maintenance activity, such as link management. Also, checks for any events, like packet reception, from any of the registered devices.
- Initialize the IPv6 Stack if built in during stack bring up.

The NETCTRL library is designed to support "potential" stack features that the user may desire within their application (e.g. DHCP server). However, the drawback of this is that the code for such features will be included in the executable even if the application never uses the features. This results in a larger footprint than is usually necessary.

To minimize this problem, the following different versions of the NETCTRL library are available in the <NDK_INSTALL_DIR>\packages\ti\ndk\netctrl\lib directory:

- **netctrl_min.** This minimal library enables only the DHCP client. It should be used when a minimal footprint is desired.
- **netctrl.** This "standard" version of the NETCTRL library enables the following features and has a medium footprint:
  - Telnet server
  - HTTP server
  - DHCP client
- **netctrl_full.** This "full" library enables all supported NETCTRL features, which include:
  - Telnet server
  - HTTP server
  - NAT server
  - DHCP client
  - DHCP server
  - DNS server

All versions of NETCTRL support NIMU, VLAN, and Raw Ethernet Socket. Each of these NETCTRL library versions is built for both pure IPv4 as well as IPv6.

If you configure the NDK in CCS with the XGCONF configuration tool, the appropriate NETCTRL library is automatically selected based on the modules you enable.

You can rebuild the NETCTRL library to include only features you want to use. To do this, edit the package.bld file in the<NDK_INSTALL_DIR>\packages\ti\ndk\netctrl directory, and redefine any of the following options. For information about rebuilding the NDK libraries, see the Rebuilding the NDK Core topic in the TI Embedded Processors Wiki.

- NETSRV_ENABLE_TELNET
- NETSRV_ENABLE_HTTP
- NETSRV_ENABLE_NAT
- NETSRV_ENABLE_DHCPCLIENT
- NETSRV_ENABLE_DHCPSERVER
- NETSRV_ENABLE_DNSSERVER

1.4 NDK Programming APIs

As previously stated, the stack has been designed for optimal isolation, and so that it may seamlessly plug in to varying run-time environments. Therefore, you may be have the opportunity to use to several different programming interfaces. They are listed here in decreasing order of relevance. All of the following are described in detail in the *TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524).

1.4.1 Operating System Abstraction

The OS abstraction consists of a custom Task and Semaphore API contained in the OS adaptation layer. The STACK and NETTOOL libraries use these abstractions so that their OS use can be adjusted by adjusting the implementation of the abstraction in OS. Note that Task and Semaphore handles created by these APIs are physically SYS/BIOS Task and Semaphore objects.

1.4.2 Sockets and Stream IO API

The sockets API is primarily consists of the standard BSD socket layer API, but contains a few other useful calls. These functions are reentrant and thread safe. They appear as an extension of the standard IO supplied with the operating system, and should not conflict with any native file support functions.

1.4.3 NETTOOL Services and Support Functions

The NETTOOL library includes both network services and basic network support functions. The API to the support functions is standardized to that of Berkeley Unix where it makes sense, with some additional functions provided for custom features.

The NETTOOL services include most network protocol servers required to operate the stack as a network server or router. The API to the services is standardized and uniform across all supported services, plus services may also be invoked by using the configuration system, bypassing the NETTOOL APIs entirely.

1.4.4 Internal Stack API

You will almost never use the internal stack API (can be thought of as kernel level API). However, it is required for some types of stack maintenance, and it is called by some of the sample source code.

1.4.5 Hardware Adaptation Layer API

You will most likely never call the HAL API directly, but it is required when moving the stack to an alternate hardware platform. The HAL is described in more detail in the *TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524) and the *TMS320C6000 Network Developer's Kit (NDK) Support Package Ethernet Driver Design Guide* (SPRUFP2).

1.5 NDK Software Directory Structure

The unzipped NDK files (for example, in c:\ti\ndk_2_21_##_##), are organized into the following subdirectories.

Directory	Description
\docs	Contains NDK documentation in PDF format. Access the online help from within CCS by choosing **Help→Help Contents** from the menus.
\eclipse	Contains files needed by CCS.
\manifests	Used internally.
\packages	The top-level package repository directory. See Section 1.5.1.

1.5.1 Directories in packages\ti\ndk

The <NDK_INSTALL_DIR>\packages\ti\ndk directory contains the following subdirectories. For each library, both source files and pre-build libraries are provided.

Directory	Description
benchmarks	Contains spreadsheets with throughput and CPU load statistics from benchmark testing on the TCI6482 board.
config	Used internally. Contains packages for all the modules configured by XGCONF and used by application code.
docs	Contains Doxygen documentation for NDK internals (for advanced users only).
hal	Contains NDK driver libraries and source code. See Section 1.3.7.
inc	NDK include file directory. See Section 1.5.2.
miniPrintf	Contains small code size printf libraries and source code. See Section 1.3.6.
netctrl	Contains libraries and source code for network startup and shutdown, including special versions for various subsets of network functionality. See Section 1.3.8
nettools	Contains libraries and source code for network tools, such as DHCP, DNS, and HTTP. See Section 1.4.3.
os	Contains libraries and source code for the OS Adaptation Layer. See Section 1.3.6.
package	Used internally.
productview	Used internally by XGCONF.
rov	Used internally by the ROV debugging tool in CCS.
stack	Contains libraries and source code for the network stack. See Section 1.3.4.
tools	Contains libraries and source code for several network tools. See Section 1.5.3.
winapps	Contains client test applications for Windows® and Linux command-prompt use. Both source code and executables are provided. See Section 1.5.4.

1.5.2 NDK Include File Directory

The include file directory (<NDK_INSTALL_DIR>\packages\ti\ndk\inc) contains all the include files that can be referenced by a network application. It is necessary to include this directory in the software tools default search path, or in the search path of the CCStudio project file. The latter method is used in the example programs. The major include files are as follows:

Filename	Description
netmain.h	Master include file for applications (stacksys.h, _nettool.h, _netctrl.h)
stacksys.h	Main include file (minus the end-application oriented include files) (usertype.h, serrno.h, socket.h, osif.h, hal.h)
_netctrl.h	Includes references for the NETCTRL scheduler library
_nettool.h	Includes references for all the services in the NETTOOL library

_oskern.h	Includes kernel level OS functions declarations
_stack.h	Includes all low level STACK interface functions
serrno.h	Standard error values
socket.h	Prototypes for all file descriptor based functions
stkmain.h	Include file used by low-level modules (not for use by applications)
usertype.h	Standard types used by the stack

Additional include files are provided in the subdirectories for the HAL, NETCTRL, NETTOOLS, OS, STACK, and TOOLS libraries.

1.5.3 Tool Programs

The NDK provides several tools for various purposes. These are located in the <NDK_INSTALL_DIR>\packages\ti\ndk\tools directory.

Subdirectory	Variants	Description
\cgi	cgi	Functions for parsing embedded HTTP Common Gateway Interface (CGI) files.
\console	console, console_ipv4	Command-line based console program.
\hdlc	hdlc	High-Level Data Link Control (HDLC) client and server.
\servers	servers, servers_ipv4	Servers used for testing NDK.

1.5.4 Windows and Linux Test Utilities

The WINAPPS directory contains four very simple test applications that can used to verify the operation of the Console example program. These test applications act as network clients for TCP send, receive, and echo, and for UDP echo operations. Most of the NDK example programs contain network data servers that can communicate with these test applications. The SEND, RECV, ECHOC, and TESTUDP applications are referenced in the description of these examples that can be found in Chapter 2.

Executable versions of these test programs are provided for both Windows and Linux.

You can use the supplied makefiles to rebuild these tools using the Microsoft Visual Studio or MinGW compiler tools.

1.5.5 Example Programs

The NDK examples have been moved to the NDK Support Package (NSP), which you download separately from the NDK. The NSP contains Ethernet driver code, libraries and network examples for various TI platforms. It is meant to supplement the Network Development Kit (NDK), which consists of the platform-independent networking code. For some platforms, the Ethernet driver and network examples are provided in the SDK for that platform.

The NSP for the OMAP-L138 ARM9 and EVM6748 DSP has the following directory structure:

Directory	Description
\docs	Contains documentation-related files.
\eclipse	Contains files used internally by CCS.
\packages	Contains software
\ti\drv	Contains drivers for OMAP-L138
\ti\ndk\examples	Contains zip files of examples for several targets

Extract the files in the zip file for your target. Both COFF and ELF examples are provided for the EVM6748. The following examples are provided in each zip file:

cfgdemo	Embedded system configuration via HTTP demonstration
client	Standard IP client demonstration
helloworld	Basic stack setup demonstration

1.6 Creating CCS Projects that Use the NDK

Follow these steps to create a Code Composer Studio project that uses the NDK and NSP:

1. Choose **File > New > CCS Project** from the CCS menus.
2. In the New CCS Project dialog, type a **Project name** and select your device **Family** and **Variant**.
3. In the **Project templates and examples** area, select a template for your project. Choose one of the SYS/BIOS examples, so that your project will include a configuration file (*.cfg), which is also used to statically configure NDK modules. For a simple starting point, you might select the **Minimal** or **Typical** example.

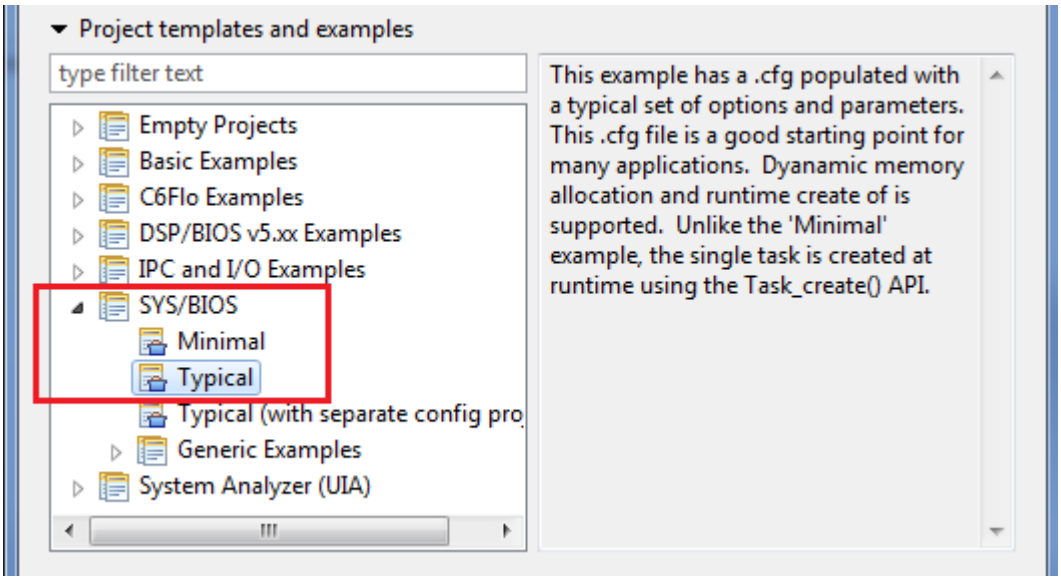


Figure 1-2. SYS/BIOS Typical Template for New CCS Project

4. Then, click **Next**.

5. On the RTSC Configuration Settings page, check the boxes next to the **NDK** and **NDK Network Support Package** products, and make sure the latest versions you have installed are selected.
6. Click on the **Platform** field. CCS will search for platform definitions that match the **Target**. Open the drop-down list of Platforms and choose your board or device.

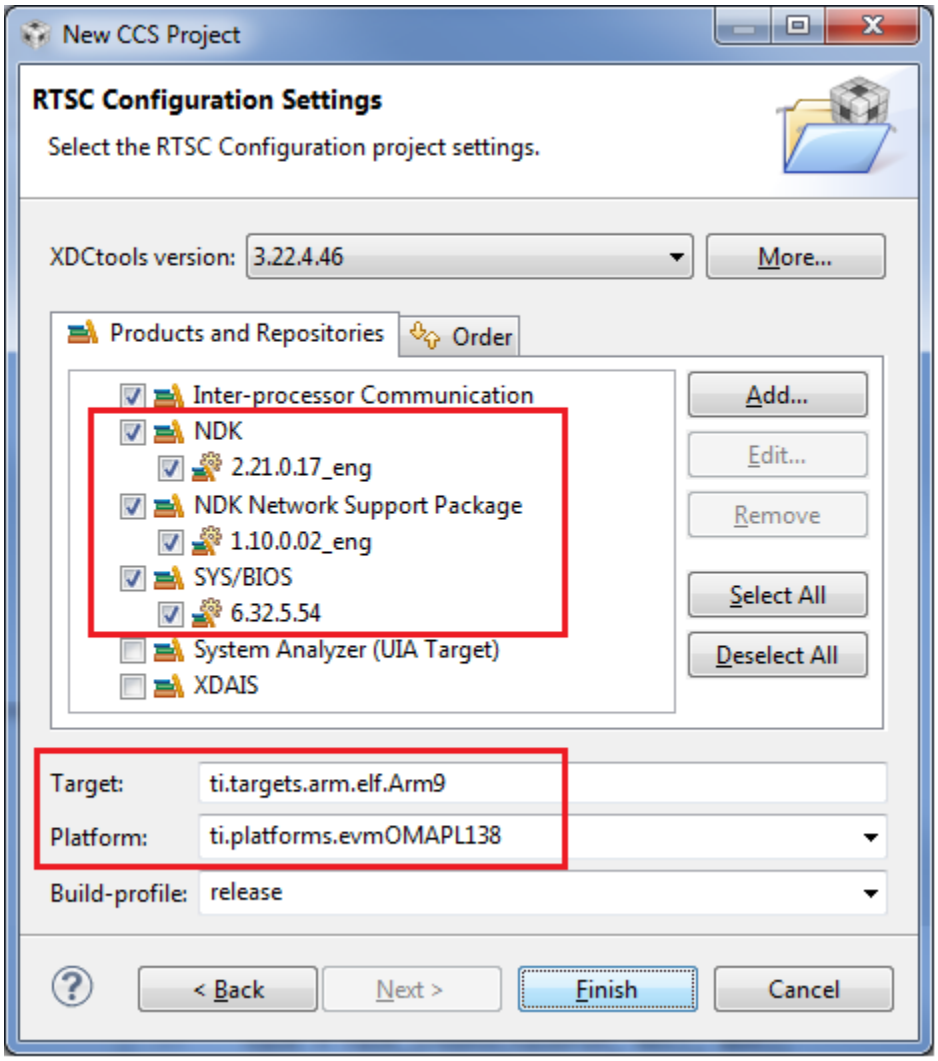


Figure 1-3. Sample RTSC Configuration Settings

7. Click **Finish** to create the CCS project.

"RTSC"—pronounced "rit-see"—stands for Real Time Software Components. It is the open-source project within Eclipse.org for reusable software packaging for use in embedded systems. XDCtools, which is installed as part of CCS, is an implementation of RTSC. You may see both "RTSC" and "XDCtools" in CCS dialogs. In general, what you need to know about XDCtools is that it provides platform definitions and tools for editing and building the static configuration file in CCS projects that use SYS/BIOS.

1.6.1 Adding NDK Support to an Existing Application

If you have an existing application that has SYS/BIOS and XDCtools support, but does not have the NDK enabled, follow these steps to enable the NDK and NSP:

1. Right-click on the CCS project in the Project Explorer view. Choose **Build Options** from the context menu.
2. Select the **General** category in the Properties dialog.
3. Go to the RTSC tab.
4. Check the boxes next to the most recently installed versions of the **NDK** and **NDK Network Support Package** products.
5. Click **OK**.
6. Reopen the *.cfg file in the project with XGCONF. The NDK and NSP should now be listed in the **Available Products** view.
7. Select the **NDK > Global** item in the **Available Products** view. You will see the Welcome sheet for NDK configuration. This sheet provides an overview of the NDK, configuration information, and documentation for the NDK.
8. Click the **System Overview** button in the Welcome sheet to see a handy diagram of the NDK modules you can configure.

1.6.2 Troubleshooting NDK Application Builds

If you get errors when attempting to build an NDK application created from scratch as described in Section 1.6, try the following:

- In the XGCONF configuration editor, make sure the NDK's Global module is enabled. There should be a green checkmark in the **Global Network Settings** box in the NDK - System Overview page. If it is not enabled, right-click on the **NDK Core Stack > Global** module in the Available Products view, and choose **Use Global** from the context menu.
- In the XGCONF configuration editor, use the Available Products view to enable the **NSP > Emac** module. Do this by right-clicking on the **Emac** module under the NSP component and choosing **Use Emac** from the context menu.
- If you needed to enable the Emac module, you should also copy the emacHooks.c file from one of the NSP examples into your project.
- Right-click on the CCS project file in the Project Explorer, and choose **Build Options**. Choose the **Build > Compiler > Include Options** category. Add the c:\ti\ndk_#_##_##_##\packages\ti\ndk\inc directory to the #include search path.

1.6.3 Creating CCS Projects for Big-Endian Applications

The NDK includes libraries built for little-endian architecture for all supported platforms. In addition, big-endian libraries are provided for the C64x+ and C66x device families. The big-endian libraries all have file extensions with an extra "e" (.a64P**e** and .ae66**e**).

The correct libraries are linked in automatically based on the architecture you choose when you create a project. It is best to specify that a CCS project should be big-endian when you create the project, rather than trying to modify the endianness setting of an existing CCS project.

When you create a Code Composer Studio project that uses the big-endian libraries, perform the following extra steps in addition to those in Section 1.6:

1. On the CCS Project page of the New CCS Project dialog, select "C6000" as the **Family**.
2. Type a filter in the **Variant** field. For example, you might type **643** to narrow the list of choices in the drop-down list.
3. Select your target from the drop-down list. For example, choose **EVMDM6437**.
4. Click the small arrow next to **Advanced settings**. This hides the "Project templates and examples" area and shows fields related to how to link and compile the project.
5. Change the **Device endianness** setting from "little" to "big".

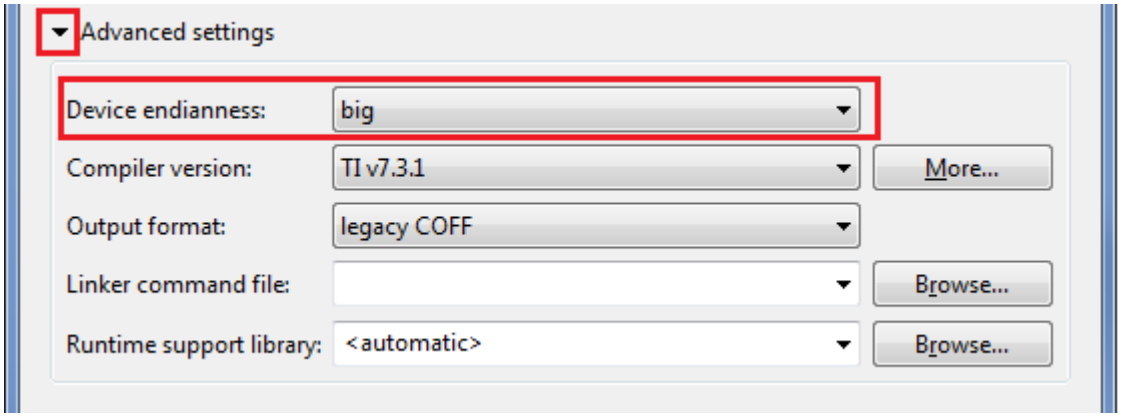


Figure 1-4. Endianness Option in Advanced Settings for New CCS Project

6. Use the small arrows to switch back to viewing the "Project templates and examples" area. Select the SYS/BIOS project template you want to use. Then, click **Next**.

7. On the RTSC Configuration Settings page of the New CCS Project dialog, notice that the **Target** for the configuration is automatically set to a big-endian target because you selected the big-endian architecture on the previous page of the New CCS Project wizard. For example, if you are using a C64x+ device, the Target may be **ti.targets.C64P_big_endian**.
8. Click on the **Platform** field. CCS will search for platforms that match the Target. Open the drop-down list of Platforms and choose your board or device. For example, **ti.platforms.evmDM6437**.

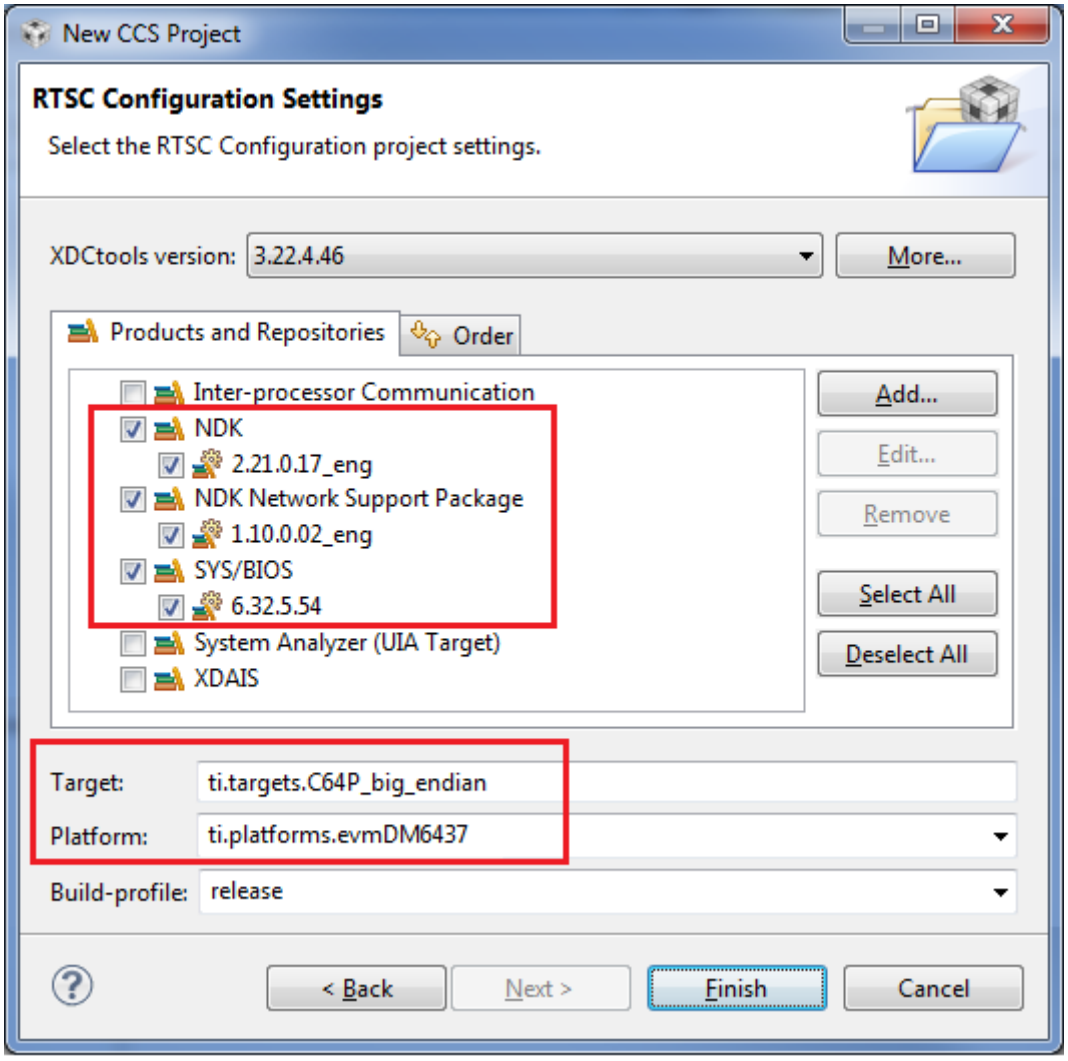


Figure 1-5. RTSC Configuration Settings for a Big-Endian Target

9. Click **Finish** to create the CCS project.

NOTE: Make sure you also link in the big-endian version of the platform support packages and ensure the hardware switch for *Big-Endian* mode is selected.

1.7 Configuring NDK Modules

To simplify configuration of the NDK and its components, the NDK now allows you to use the XGCONF configuration tool within Code Composer Studio (CCS). Graphical displays let you enable and set properties as needed, and context-sensitive help provides information about individual fields. XGCONF is the same configuration tool used to configure SYS/BIOS. The application's configuration file (*.cfg) can configure both NDK and SYS/BIOS modules and objects.

In previous versions of the NDK, applications were configured by writing C code that called CfgNew() to create a configuration database and other Cfg*() functions to add various settings to that configuration database. In addition, some configuration was done in the linker command file. (Internally, the same C code is now generated to update the same configuration database when the *.cfg file is built.)

You can still choose to use C code to set up the configuration database if you have legacy code. However, you must choose one method or the other to configure your application.

NOTE: You should not mix configuration methods. If you have legacy NDK applications that use the old C-based configuration method, you should either continue to use that method or convert the configuration entirely to an *.cfg file configuration. If a project uses both methods, there will be conflicts between the two configurations.

1.7.1 Opening the XGCONF Configuration Editor

When you create a project using a SYS/BIOS template, the project will contain a configuration file (*.cfg) that can be edited with the XGCONF graphical editor in CCS. If you checked the boxes to enable use of the NDK and NSP when you created the project, you can configure your application's use of the NDK modules. The configuration file is processed during the build to generate code that configures your application.

This section provides an overview of how to use the XGCONF graphical editor. For more details, see Section 2.2 of the *TI SYS/BIOS Real-time Operating System User's Guide* (SPRUEX3)

To open XGCONF, follow these steps:

1. Make sure you are in the **CCS Edit** perspective of CCS. If you are not in that perspective, click the **CCS Edit** icon to switch back.

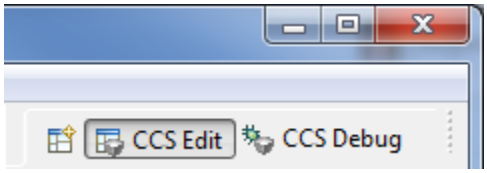


Figure 1-6. C/C++ Perspective Icon

2. Double-click on the *.cfg configuration file in the Project Explorer tree. While XGCONF is opening, the CCS status bar shows that the configuration is being processed and validated.
3. When XGCONF opens, you see the Welcome sheet for SYS/BIOS. You should see categories for the **NDK Core Stack** and your NSP in the Available Products area. If you do not, your CCS Project does not have NDK support enabled. See Section 1.6.1 to correct this problem. (If the configuration is shown in a text editor instead of XGCONF, right-click on the .cfg file in the Project Explorer and choose **Open With > XGCONF**.)

4. Click the **Global** item in either the **Available Products** view (under the NDK Core Stack category) or in the **Outline** view

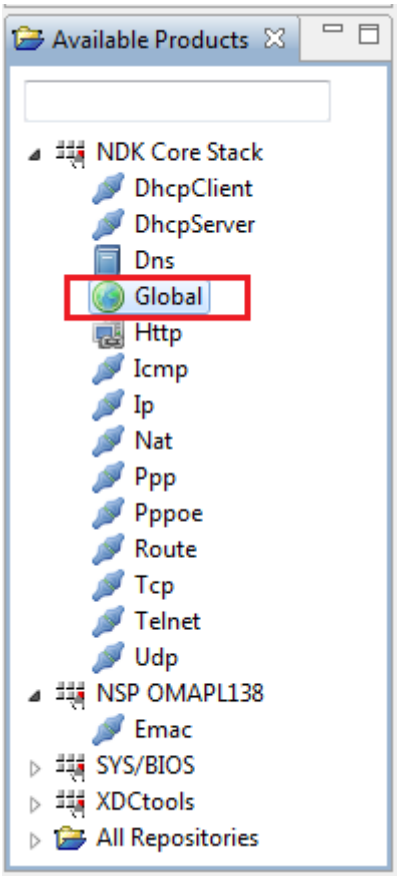


Figure 1-7. NDK Modules in Available Products List

5. You will see the Welcome sheet for NDK configuration. This sheet provides an overview of the NDK, configuration information, and documentation for the NDK.

6. Click the **System Overview** button to see a handy diagram of the NDK modules you can configure. If you are editing the configuration of one of the NSP examples, notice the green checkmarks next to some modules. These checkmarks indicate that support for the modules have been enabled in the configuration. (If you created a new NDK project as described in Section 1.6, only the Global module is enabled by default.)

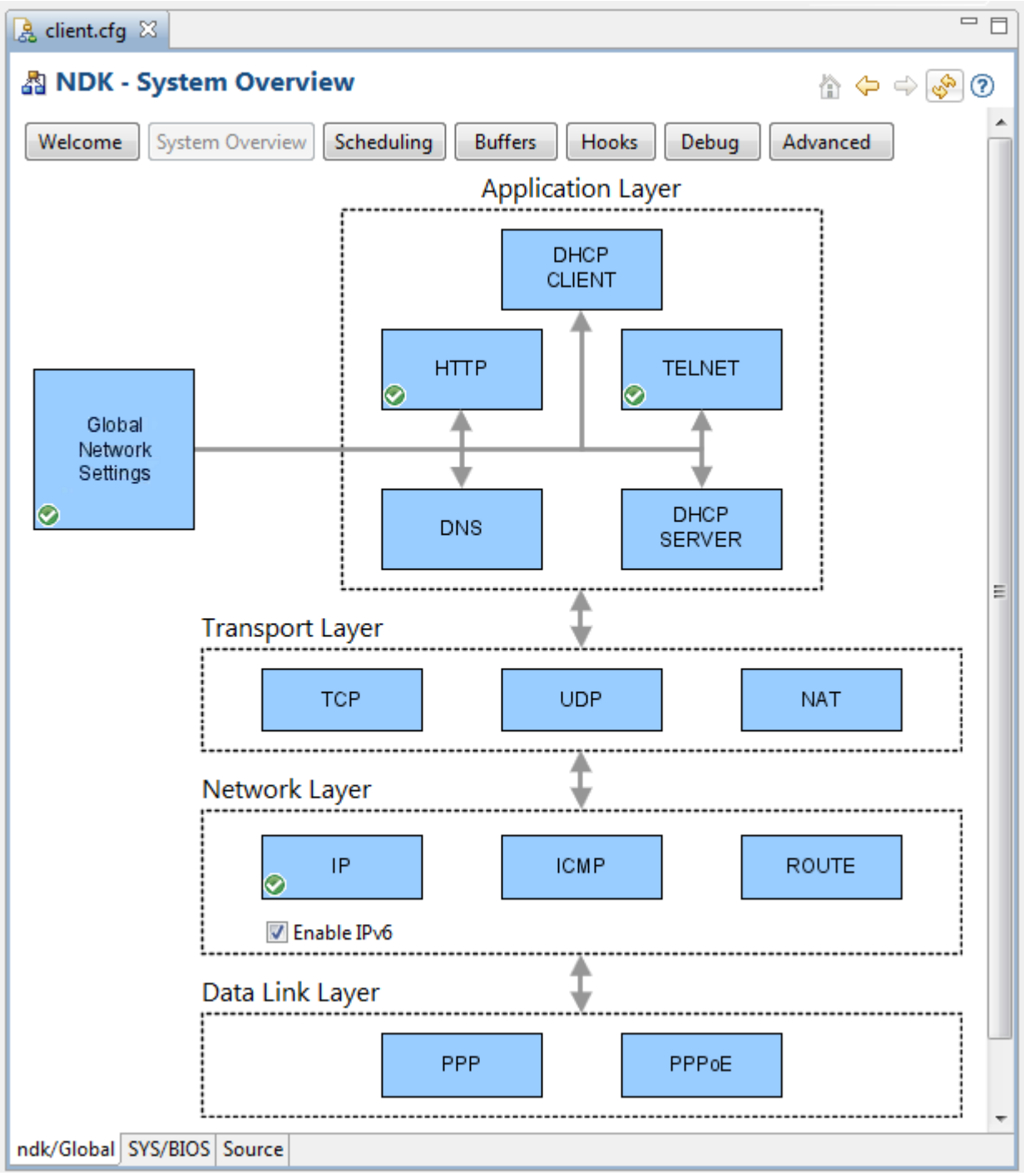


Figure 1-8. NDK System Overview Diagram

1.7.2 Adding a Module to Your Configuration

To add support for a module to your configuration, follow these steps:

1. Click on a module that you want to use in your application in the **System Overview** diagram or in the **Available Products** view.
2. In the Module Settings sheet, check the box to **Add the <module> to my configuration**. (You can also right-click on a module in the **Available Products** view and choose **Use <module>** from the context menu.)

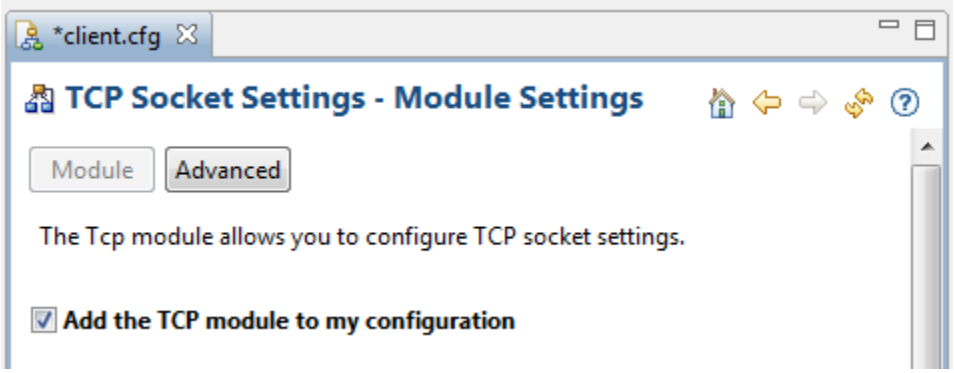


Figure 1-9. Adding a Module to the Configuration

3. Notice that the module you added to the configuration is now listed in the **Outline** view.

1.7.3 Setting Properties for a Module

To set properties for a module, go to the Module Settings sheet and type or select the settings you want to use.

If you want information about a property, point to the field with your mouse cursor.

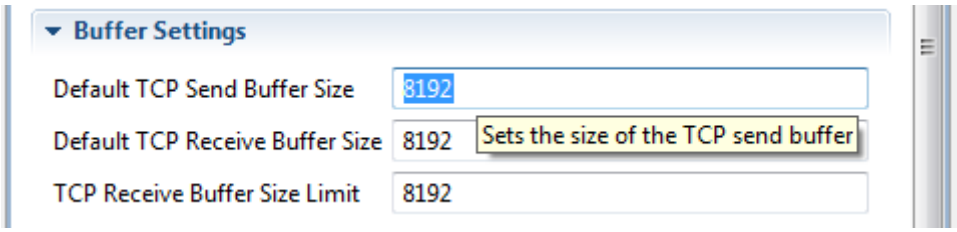


Figure 1-10. Module Properties

For details about properties, right-click and choose **Help** from the context menu. This opens the CDOC online reference system for the NDK. The properties names listed in this online help are the names used in the configuration source file. You can click the **Source** tab at the bottom of the XGCONF editor window to see the actual source statements.

1.7.4 Adding an Instance for a Module

Some of the NDK modules allow you to create instances of that type. For example, you can create instances of DHCP servers, DNS servers, HTTP servers, NAT servers, and Telnet servers. To create such instances, follow these steps:

1. Go to the property sheet for the module for which you will add an instance.
2. Click the **Instance** button at the top of the Module Settings sheet.
3. Click the **Add** button to open a property window for a new instance. You can set properties here or in the Instance Settings sheet.
4. Click **OK** to create the instance.

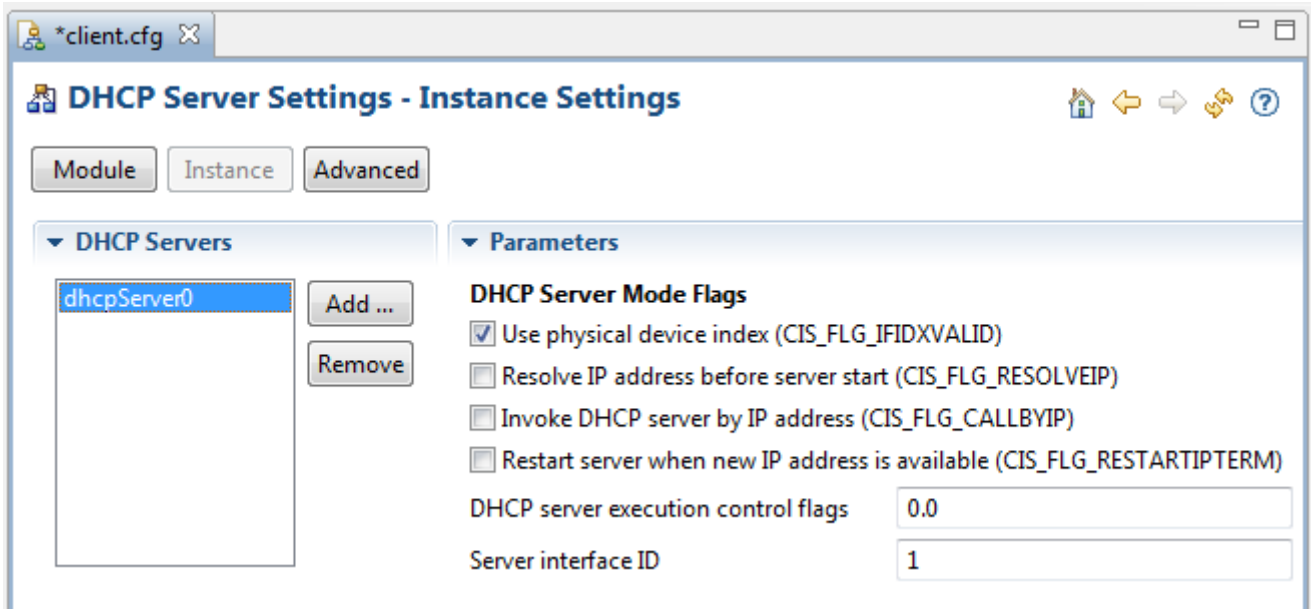


Figure 1-11. DHCP Server Instance

5. Notice that the instance you created is also listed in the **Outline** view.

1.7.5 Saving Changes to the Configuration

To save changes to your configuration, press Ctrl+S. You can also choose **File > Save** from the CCS menus.

When you make changes to the configuration or save the configuration, your settings are validated. Any errors or warnings found are listed in the **Problems** view and icons in the **Outline** view identify any modules or instances that have problems.

Example Applications

This section describes the main example applications included with the NSP software. The example applications are designed to provide a small sample of potential applications that can be developed with the NDK.

NOTE: The NDK examples described in this chapter have been moved to the NDK Support Package (NSP), which you must download separately from the NDK. The NSP is meant to supplement the Network Development Kit (NDK), which consists of the platform-independent networking code. For some platforms, the NSP is provided in the SDK for that platform.

The descriptions in this chapter sometimes refer specifically to the NSP that supports the EVM6748 and the OMAP-L138 and ARM9.

See Section 1.5.5 for a description of the directory structure of the examples in the NSP. You will need to unzip the examples for your target in order to use or examine them.

The sample applications in the NSP can be run as is for a quick demonstration, but it is recommended to use these samples as sample source code in developing new applications. For this, a working knowledge of how the Code Composer Studio environment interacts with the NDK is helpful. Chapter 3 of this user's guide is dedicated to the development of networking applications using CCStudio.

Topic **Page**

2.1 Troubleshooting

Some of the example applications described in this section require a network with support for DHCP. If DHCP is not available, only the Configuration example can be run as-is. The remaining examples can be rebuilt to use a fixed IP configuration using Code Composer Studio. See Chapter 3 for details on network application initialization.

On some platforms, it is necessary to reset the device before loading an OUT file. If the example file fails to initialize properly, it can be stopped or sent off into the weeds. This is caused by cache and interrupts being in non-default state when loading.

The example program for each individual platform is pre-set with the preferred cache configuration. When internal memory is not required, 4-way cache is used. Otherwise, cache is selected to meet internal memory requirements. Any system that requires a specific memory/cache map should be clearly documented.

2.2 The Network Client Example Application

NOTE: The example described in this section has been moved to the NDK Support Package (NSP), which you must download separately from the NDK. Examples are stored in zip files in the <NSP_INSTALL_DIR>\packages\ti\ndk\examples directory of the NSP installation.

The client example is the most important of all the example programs since it includes the most components of an actual network application. The client example can use either DHCP or a statically configured IP address. It launches a console application accessible via Telnet, an HTTP server with a couple of example WEB pages, plus several data servers that can be tested by running client test applications on a Windows PC. This application also illustrates the IPv6 stack functionality.

2.2.1 Building the Application

The application can be rebuilt directly from its project file using Code Composer Studio™.

2.2.2 Loading the Application

The application is loaded and executed via Code Composer Studio. It is a good idea to reset the board before loading CCStudio, but this should not be required. The application displays status messages in CCStudio's standard IO output window (Stdout).

On a successful execution, one of the status lines printed by the application displays the client's IP address (either through DHCP or static configuration). Once this address is displayed, the DSP or ARM responds to requests made to its IP address. When using DHCP, it is possible that the application will be unable to obtain an IP address from a DHCP server. If so, the application eventually prints a DHCP status message with the fault condition. Note that all the messages are generated by the main client module in `CLIENT.C`.

2.2.3 Testing the Application

Once the application is executing and has printed out its IP address, several tests can be performed.

2.2.3.1 HTTP Server

To see the HTTP server in action, run an Internet browser, and point it to the IP address displayed by the application. If the client application's IP address is 196.12.1.14, the URL would be:

```
http://196.12.1.14
```

Be sure and disable any proxy settings on the browser if your network is behind a firewall.

The browser displays a small WEB page describing the example application. There are server status screens that can be accessed off this page. The source code used to generate these pages is further described in the HTTP appendix of the *TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524).

2.2.3.2 Telnet Server

The client example application also includes a console application with several tests and status query functions available. In order to get to the console, simply telnet into the application's IP address. Note that the console program will timeout and disconnect after a period of inactivity.

To get a list of console commands, type *help* or simply *?*. This action prints a list of console commands to the telnet terminal. The console program is important as a programming demonstration as much as a run time demonstration. There are many functions in the console program that display or test features particular to the NDK. When an application developer wants to use these features in their application, the console example source code can be useful as a guide.

2.2.3.3 Data Servers

To try out the data servers, use the Windows or Linux test applications found in the WINAPPS directory off the NDK root. The applications are command line driven and require a target IP address. For example, type:

```
send 196.12.1.14
```

to start the data receiver. This requests data from the server running on the DSP or ARM. To get more accurate benchmark numbers, the number of display updates can be reduced by typing an update period. For example:

```
recv 196.12.1.14 100
```

starts the data send test (receive from the DSP or ARM's point of view) with a display update interval of 100 iterations.

```
echoc 196.12.1.14 100
```

starts the TCP data echo test (echoes back the characters it receives from the DSP or ARM) with a display update interval of 100 iterations.

```
testudp 196.12.1.14
```

starts an UDP data server that tests the UDP client running on the DSP or ARM.

All the Windows test clients run until a key is pressed, or Control-C in the event of an error (for instance, trying to connect to a bad IP address).

2.2.3.4 IPv6 Stack Testing

The IPv6 stack comes with various command line driven tools like Ping6 (IPv6 based Ping utility), Tftp6 (IPv6 based TFTP client), and various test IPv6 socket-based applications, etc. Before getting to test these applications, it is required by the user to first setup IPv6 address on the desired interface. This section describes the steps to follow to set up and use any of the IPv6 utilities.

1. Telnet into the application's IP address from a PC to get to the console where the IPv6 commands can be entered.
2. Type help or ? at the prompt to get a list of commands supported. If the IPv6 stack was successfully compiled into the application, the following command line options must be seen on the screen:

ipv6	IPv6 Configuration
ping6	Test echo request over IPv6
v6nslookup	Lookup hostname or IPv6 address

3. Initialize and attach the IPv6 stack and setup a default link local address on a desired interface, say interface name eth0. To do so, type:

   ipv6 init eth0

   This sets up a link local IPv6 address on eth0 and the interface is ready for IPv6 communication.

4. Type the following to get an additional list of commands supported under ipv6:

   ipv6

   The following commands will be presented:

   [ipv6 Command]

   Use this to configure/display the IPv6 stack properties.

ipv6	Displays the usage screen
ipv6 init <if>	Initialize the IPv6 stack on an interface
ipv6 deinit <if>	Deinitialize the IPv6 stack on an interface
ipv6 add <if> <IPAddr> <NumBits> <VLT> <PLT> <ANYCAST/UNICAST>	Adds an IPv6 address
ipv6 del <if> <IPAddr>	Deletes an IPv6 address
ipv6 neigh	Display the neighbor table
ipv6 route	Display the IPv6 routing table
ipv6 bind	Display a list of all configured IPv6 addresses
ipv6 stats	Displays IPv6 stack statistics for core IPv6, TCP, UDP, RAW, ICMPv6 modules
ipv6 test	Test commands used to test the IPv6 API Type ipv6 test to get a list of all the test commands available

   These commands can be used to configure any manual global addresses, check various IPv6 stats, and run various tests on IPv6 stack using IPV6 sockets.

5. Type the following to ping another IPv6 device with say a link local IPv6 address fe80::213:72ff:fe8b:234b :-

   ping6 fe80::213:72ff:fe8b:234b 1

   The last parameter, i.e., interface index (scope id), must be specified only if using link local addresses for ping. If using global IPv6 addresses, the scope id need not be specified.

6. Type the following to test TFTP6 client:

   ipv6 test tftp <SrvIP> <File> [scopeid]

   The scope id is required again only if the server IP address being used is a link local address.

7. Use v6nslookup command to resolve a given hostname to IPv6 address and vice-a-versa.

8. Type the following to bring down the IPv6 stack on a given interface, say eth0:

   ipv6 deinit eth0

   All IPv6 addresses are removed from this interface and the interface can no longer be used for IPv6 communication.

   For more details on the IPv6 Stack APIs and data structures, refer to the IP Version 6 (IPv6) Stack API section of the *TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524).

2.3 The Network Configuration Example Application

NOTE: The example described in this section has been moved to the NDK Support Package (NSP), which you must download separately from the NDK. Examples are stored in zip files in the <NSP_INSTALL_DIR>\packages\ti\ndk\examples directory of the NSP installation.

The Configuration Demo (CFGDEMO) example illustrates how the stack running in an embedded environment can be easily configured without relying on DHCP. The demo boots up the DSP or ARM in an idle state with no IP address. You assign a temporary IP address, and then an HTTP client browser completes the configuration.

2.3.1 Building the Application

The application can be rebuilt directly from its project file using Code Composer Studio.

2.3.2 Loading the Application

The application is loaded and executed via Code Composer Studio. The application displays status messages in CCStudio's standard IO output window (Stdout).

On a successful execution, one of the status lines printed by the application displays *GetIP Ready*. This indicates that the DSP or ARM board is ready to have an IP address assigned by you. Note that all the messages are generated by the main module in `CFGDEMO.C`.

2.3.3 Configuring the Application

Once the application is executing and has printed out its GetIP Ready message, it is ready for configuration.

2.3.3.1 Setting the Initial IP Address

The first step in configuring the device is to assign it a temporary (or permanent) IP address. The CFGDEMO application uses the ICMP ping message to initially detect its IP address.

Once a free IP address is chosen (say 192.63.10.5), you can assign the IP address to the DSP or ARM by using the ping command from another machine. Note that the DSK does not reply to ARP requests when not configured; therefore, the MAC address for the chosen IP must be manually entered.

For those devices requiring a daughtercard, the MAC address of the DSK is usually found on the white label affixed to the Ethernet daughtercard. For example, if the MAC address were 08-00-28-32-08-26, to assign this MAC address to the selected IP address, on a Windows command line, type:

```
arp -s 192.63.10.5
                08-00-28-32-08-26
```

Next, to assign the IP address to the CFGDEMO application on the DSK, type

```
ping 192.63.10.5
```

The DSK board should start replying to the ping command. Since the demo application prints some additional status messages to CCStudio, it may miss a ping request during this time.

Once the application is responding to ping requests, it is ready for full configuration.

2.3.3.2 Full System Configuration

To complete the system configuration, an Internet browser is used. Run the browser and point it to the IP address assigned to the DSP or ARM in the previous step. If the IP address is 192.63.10.5, the URL would be:

```
http://192.63.10.5
```

Be sure and disable any proxy settings on the browser if your network is behind a firewall.

The browser displays a small WEB page describing the example application. There is a button on this page that takes you to the configuration page. The password required to enter the configuration page is printed on the screen.

Once in the configuration page, simply fill out the form and press the Submit button.

If DHCP was selected on the configuration form, the application attempts to get an IP address from a DHCP server as with the Client example described in the previous section.

2.3.4 Testing the Application

Once the application is configured, has restarted and printed out its IP address, several tests can be performed. These tests are identical to those in the previous Client example.

2.3.4.1 Telnet Server

The example application includes a console application with several tests and status query functions available. In order to get to the console, simply telnet into the application's IP address. Note that the console program will timeout and disconnect after a period of inactivity. Note also that the Telnet console can be disabled from the configuration WEB page.

2.3.4.2 Data Servers

To try out the data servers, use the Windows test applications found in the \WINAPPS directory. The applications are command line driven and require a target IP address. For example, type:

```
recv 192.63.10.5
```

to start the data receiver. This action requests data from the server running on the DSP or ARM. To get more accurate benchmark numbers, the number of display updates can be reduced by typing an update period. For example:

```
send 192.63.10.5 100
```

starts the data send test (receive from the DSP or ARM's point of view) with a display update interval of 100 iterations.

```
echoc 192.63.10.5 100
```

starts the TCP data echo test (echoes back the characters it receives from the DSP or ARM) with a display update interval of 100 iterations.

```
testudp 196.12.1.14
```

starts an UDP data server that tests the UDP client running on the DSP or ARM.

All the Windows test clients run until a key is pressed, or Control-C in the event of an error (for instance, trying to connect to a bad IP address).

2.4 The Network HelloWorld Example Application

NOTE: The example described in this section has been moved to the NDK Support Package (NSP), which you must download separately from the NDK. Examples are stored in zip files in the <NSP_INSTALL_DIR>\packages\ti\ndk\examples directory of the NSP installation.

The helloWorld example is a skeleton application intended to provide the application programmer with a basic stack setup, to which you can add your code.

2.4.1 Building the Application

The client example is located in the `EXAMPLE\NETWORK\HELLOWORLD` directory off the NDK root. The application can be rebuilt directly from its project file using Code Composer Studio.

2.4.2 Loading the Application

The application is loaded and executed using Code Composer Studio. The application displays status messages in CCStudio's standard IO output window (Stdout).

On a successful execution, one of the status lines printed by the application displays the client's IP address (either through DHCP or static configuration). Once this address is displayed, the DSP or ARM responds to requests made to its IP address. When using DHCP, it is possible that the application will be unable to obtain an IP address from a DHCP server. If so, the application eventually prints a DHCP status message with the fault condition. Note that all the messages are generated by the main client module in `HELLOWORLD.C`.

2.4.3 Testing the Application

Once the application is executing and has printed out its IP address, several tests can be performed.

2.4.3.1 HelloWorld

To try out the example, use the Windows test application found in the WINAPPS directory of the NDK root. The application is command driven and requires a target IP address, such as:

helloWorld 192.63.10.5

This sends Hello World! through a UDP socket connection, and reads the transmitted information by the stack.

2.5 The Serial Client Examples

The serial client and serial router examples are no longer provided with the NDK and NSP.

Network Application Development

Developing a network application with the NDK software is as easy as programming with a standard sockets API. However, integrating with Code Composer Studio, SYS/BIOS, and system initialization may be unfamiliar. This chapter describes how to begin developing network applications. It discusses the issues and guidelines involved in the development of network applications using the NDK libraries.

Topic **Page**

3.1 Configuring the NDK with XGCONF

To make configuration easier than in previous versions of the NDK, you can now use the XGCONF configuration tool within CCS to configure use of the NDK. Graphical displays let you enable and set properties as needed. XGCONF is the same configuration tool used to configure SYS/BIOS. The same configuration in one project can configure both NDK and SYS/BIOS modules and objects.

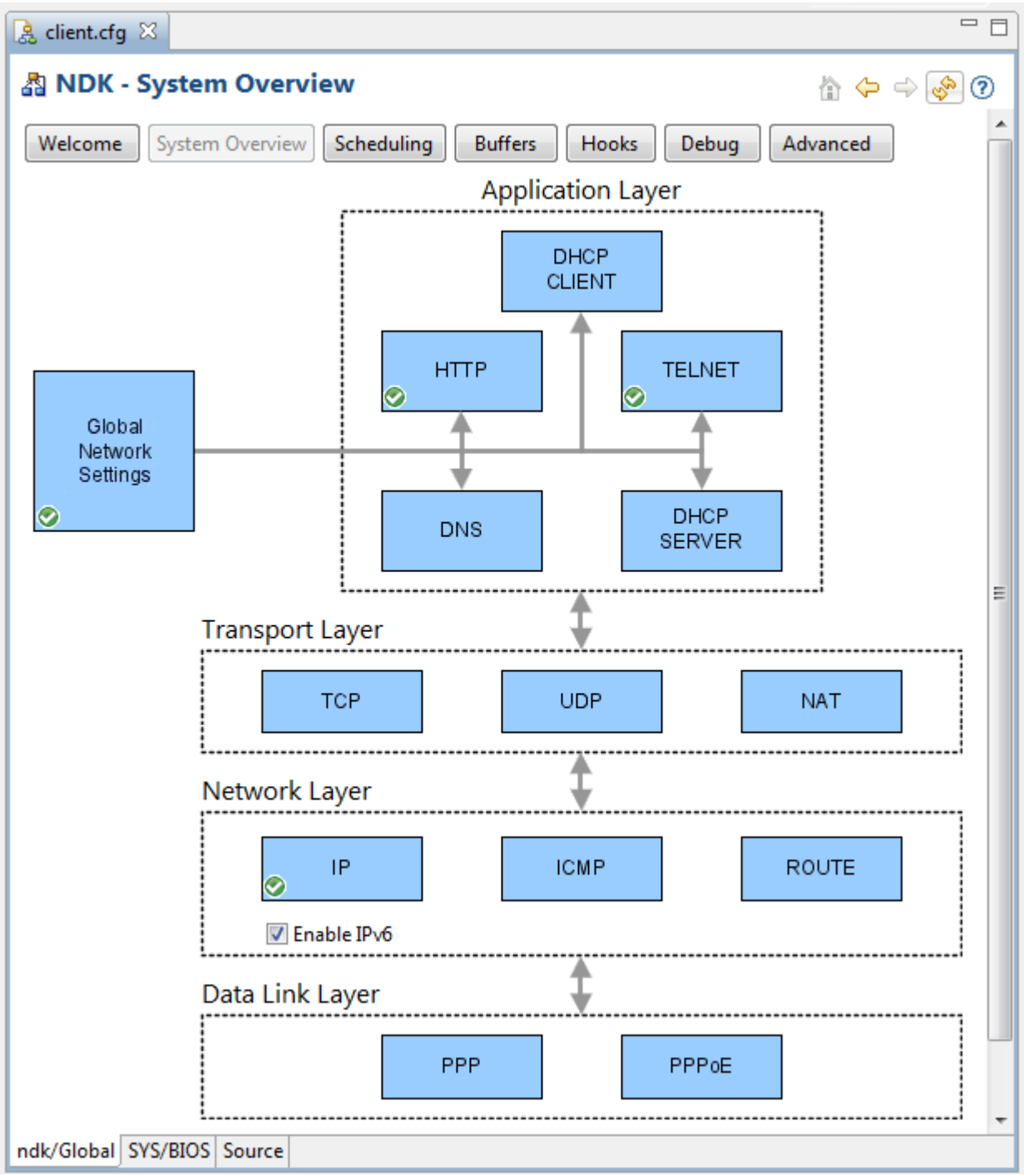
We recommend using XGCONF for configuration of NDK applications. If you use this method, many development tasks that were previously required are automated.

When you create a project using a SYS/BIOS template, the project will contain a configuration file (*.cfg) that can be edited with the XGCONF graphical editor in CCS. If you checked the boxes to enable use of the NDK and NSP when you created the project, you can configure your application's use of the NDK modules. The configuration file is processed during the build to generate code that configures your application.

In previous versions of the NDK, applications were configured by writing C code that called CfgNew() to create a configuration database and other Cfg*() functions to add various settings to that configuration database. In addition, some configuration was done in the linker command file. NDK v2.21 adds the ability to configure the NDK modules through the XGCONF configuration editor. (Internally, the same configuration database is updated when the *.cfg file is built.) Internally, the XGCONF configuration generates C code that calls into and updates the configuration database used in previous versions of the NDK. In fact, you can still choose to use the configuration database if you have legacy code. However, you must choose one method or the other to configure your application.

NOTE: You should *not* mix configuration methods. If you have legacy NDK applications that use the old C-based configuration method, you should either continue to use that method or convert the configuration entirely to an *.cfg file configuration. If a project uses both methods, there will be unpredictable conflicts between the two configurations.

1. To open XGCONF, simply double-click the *.cfg file in your application's project. See the steps in Section 1.7 for how to use XGCONF with the NDK. For more details, see Chapter 2 of the *TI SYS/BIOS Real-time Operating System User's Guide* (SPRUEX3).
2. When XGCONF opens, you see the Welcome sheet for SYS/BIOS. You should see categories for the **NDK Core Stack** and your NSP in the Available Products area. If you do not, your CCS Project does not have NDK support enabled. See Section 1.6.1 to correct this problem. (If the configuration is shown in a text editor instead of XGCONF, right-click on the .cfg file in the Project Explorer and choose **Open With > XGCONF**.)
3. Click the **Global** item in either the **Available Products** view (under the NDK Core Stack category) or in the **Outline** view
4. You see the Welcome sheet for NDK configuration. This sheet provides an overview of the NDK, configuration information, and documentation for the NDK.
5. Click the **System Overview** button to see a handy diagram of the NDK modules you can configure. If you are editing the configuration of one of the NSP examples, notice the green checkmarks next to some modules. These checkmarks indicate that support for the modules have been enabled in the configuration. (If you created a new NDK project as described in Section 1.6, only the Global module is enabled by default.)



The XGCONF configuration automatically performs the following actions for you:

- Generates C code to create and populate a configuration database.
- Generates C code to act as the network scheduling function and to perform network activity.

The following C functions are generated as a result of using the NDK Global module for configuration. You should take care not to write application functions with these names.

- *ti_ndk_config_Global_stackThread()*: The NDK stack thread function.
- *NetworkOpen()*: function that is called automatically by *NC_NetStart()*.
- *NetworkClose()*: function that is called automatically by *NC_NetStart()*.
- *NetworkIPAddr()*: function that is called automatically by *NC_NetStart()*.
- *ti_ndk_config_Global_serviceReport()*: Service report callback function.

3.1.1 Linked Libraries Configuration

The **Global** module is required in NDK applications. It determines which core libraries and which flavor of the NDK stack library are linked with your application. By default, it is also used to configure global stack settings and generate NDK configuration code. The following libraries are linked in by default via the Global module:

- stack
- cgi
- console
- hdlc
- miniPrintf
- netctrl
- nettool
- os
- servers

In addition, the appropriate version of the stack library (stk*) is linked in depending on whether you enable the NAT, PPP, PPPoE modules in your configuration.

Click the **System Overview** button in Global NDK sheet. Notice that if you have the IP module enabled, you can check or uncheck the **Enable IPv6** box. This setting controls whether the application is linked with libraries that support IPv4 or IPv6 when you build the application.

3.1.2 Global Scheduling Configuration

In addition to the **Welcome** tab that described the NDK and the **System Overview** tab that provides a diagram of the NDK modules in use, the Global module also provides several tabs that let you set various configuration options for the NDK stack. The next tab is the **Scheduling** tab, which lets you control how Task threads are scheduled and how much stack memory they can use.

3.1.2.1 Network Scheduler Task Options

You can configure the Network Scheduler Task Priority with XGCONF by selecting the NDK's **Global** module and then clicking the **Scheduling** button.

Network Scheduler Task Priority is set to either Low Priority (NC_PRIORITY_LOW) or High Priority (NC_PRIORITY_HIGH), and determines the scheduler Task's priority relative to other networking Tasks in the system.

3.1.2.2 Priority Levels for Network Tasks

The stack is designed to be flexible, and has a OS adaptation layer that can be adjusted to support any system software environment that is built on top of SYS/BIOS. Although the environment can be adjusted to suit any need by adjusting the HAL, NETCTRL and OS modules, the following restrictions should be noted for the most common environments:

1. The Network Control Module (NETCTRL) contains a network scheduler thread that schedules the processing of network events. The scheduler thread can run at any priority with the proper adjustment. Typically, the scheduler priority is low (lower than any network Task), or high (higher than any network Task). Running the scheduler thread at a low priority places certain restrictions on how a Task can operate at the socket layer. For example:
   - If a Task polls for data using the recv() function in a non-block mode, no data is ever received because the application never blocks to allow the scheduler to process incoming packets.
   - If a Task calls send() in a loop using UDP, and the destination IP address is not in the ARP table, the UDP packets are not sent because the scheduler thread is never allowed to run to process the ARP reply.
     These cases are seen more in UDP operation than in TCP. To make the TCP/IP behave more like a standard socket environment for UDP, the priority of the scheduler thread can be set to high priority. See Chapter 4 for more details on network event scheduling.
2. The NDK requires a re-entrance exclusion methodology to call into internal stack functions. This is called kernel mode by the NDK, and is entered by calling the function *llEnter()* and exited via *llExit()*. Application programmers do not typically call these functions, but you must be aware of how the functions work.

   By default, priority inversion is used to implement the kernel exclusion methods. When in kernel mode, a Task's priority is raised to OS_TASKPRIKERN. Application programmers need to be careful not to call stack functions from threads with a priority equal to or above that of OS_TASKPRIKERN, as this could cause illegal reentrancy into the stack's kernel functions. For systems that cannot tolerate priority restrictions, the NDK can be adjusted to use Semaphores for kernel exclusion. This can be done by altering the OS adaptation layer as discussed in Section 5.2.3, or by using the Semaphore based version of the OS library: OS_SEM.

3.1.2.2.1 Stack Sizes for Network Tasks

Care should be taken when choosing a Task stack size. Due to its recursive nature, a Task tends to consume a significant amount of stack. A stack size of 3072 is appropriate for UDP based communications. For TCP, 4096 should be used as a minimum, with 5120 being chosen for protocol servers. The thread that calls the NETCTRL library functions should have a stack size of at least 4096 bytes. If lesser values are used, stack overflow conditions may occur.

3.1.2.3 Priorities for Tasks that Use NDK Functions

In general, Tasks that use functions in the network stack should be of a priority no less than OS_TASKPRILOW, and no higher than OS_TASKPRIHIGH. For a typical Task, use a priority of OS_TASKPRINORM. The values for these #define variables can be changed with XGCONF by selecting the NDK's **Global** module and then clicking the **Scheduling** button.

In addition, Task priorities can be altered by adjusting the OSENVCFG structure as described in the *TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524); however, this is strongly discouraged. When altering the priority band, care must be taken to account for both the network scheduler thread and the kernel priority.

3.1.3 Global Buffer Configuration

You can configure the buffers used by the NDK by selecting the NDK's **Global** module and then clicking the **Buffers** button. This page lets you configure the sizes and locations of the NDK Packet Buffer Manager (PBM) and the Memory Manager Buffer.

The NDK defines some special memory segments via the pragma:

```
#pragma DATA_SECTION( memory_label,
             "SECTIONNAME" )
```

The NDK sections are defined by default as subsections of the .far memory segment. External memory is usually used for the .far section. The additional section names are shown below.

.far:NDK_PACKETMEM— This section is defined in the HAL and OS adaptation layers for packet buffer memory. The size required is normally 32k bytes to 48k bytes. You can configure this buffer with XGCONF by selecting the NDK's **Global** module and then clicking the **Buffers** button.

.far:NDK_MMBUFFER— This section is defined by the memory allocation system for use as a scratchpad memory resource. The size of the memory declared in this section is adjustable, but the default is less than 48k bytes. You can configure this buffer with XGCONF by selecting the NDK's **Global** module and then clicking the **Buffers** button.

.far:NDK_OBJMEM— This section is a catch-all for other large data buffer declarations. It is used by the example application code and the OS adaptation layer (for print buffers).

You can use the Program.sectMap[] configuration array to configure section placement. For details about controlling the placement of sections in memory, see Chapter 6 on Memory in the *TI SYS/BIOS Real-time Operating System User's Guide* (SPRUEX3).

The Memory Allocation Support section of the *TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524) describes the memory allocation API provided by the OS library for use by the various stack libraries. Although the stack's memory allocation API has some benefits (it is portable, bucket based to prevent fragmentation, and tracks memory leaks), the application code is expected to use the standard malloc()/free() or equivalent Memory module allocation routines provided by SYS/BIOS.

3.1.4 Global Hook Configuration

You can configure callback (hook) functions by selecting the NDK's **Global** module and then clicking the **Hooks** button. You can specify functions to be called at the following times:

- **Stack Thread Begin.** Runs at the beginning of the generated *ti_ndk_config_Global_stackThread()* function, before the call to *NC_SystemOpen()*. Note that no NDK-related code can run in this hook function because the *NC_SystemOpen()* function has not yet run.
- **Stack Thread Initialization.** Runs in the *ti_ndk_config_Global_stackThread()* function, immediately after the function call to create a new configuration, *CfgNew()*.
- **Stack Thread Delete.** Runs in the *ti_ndk_config_Global_stackThread()* function, immediately after exiting from the while() loop that calls *NC_NetStart()*, but before the calls to *CfgFree()* and *NC_SystemClose()*. (Configuration database calls, such as *CfgNew()*, are still made internally even if you use the XGCONF configuration method. These calls are described in Section 3.2.4, but generally you do not need to be concerned with them if you are using XGCONF for configuration.)
- **Status Report.** Runs at the beginning of the generated *ti_ndk_config_Global_serviceReport()* function.
- **Network Open.** Runs at the beginning of the generated *NetworkOpen()* function, when the stack is ready to begin creating application supplied network Tasks. Note that this function is called during the early stages of stack startup, and must return in order for the stack to resume operations.
- **Network Close.** Runs at the beginning of the generated *NetworkClose()* function, when the stack is about to shut down.
- **Network IP Address.** Runs at the beginning of the generated *NetworkIPAddr()* function, when an IP address is added to or removed from the system.

Hook functions must be defined using the following format:

```
Void functionName(Void)
```

If you specify a hook function in the configuration, but do not define the function in your C code, a linker error will result.

For example, the following function could be called as the **Stack Thread Initialization** hook function to open an SMTP server application:

```
static SMTP_Handle hSMTP;

//
// SmtpStart
// This function is called after the configuration has been loaded
//
static void SmtpStart( )
{
    // Create an SMTP server Task
    hSMTP = SMTP_open( );
}
```

The above code launches a self-contained application that needs no further monitoring, but the application must be shut down when the system shuts down. This is done via the **Stack Thread Delete** callback function.

```
//
// SmtpStop
// This function is called when the network is shutting down
//
static void SmtpStop()
{
    // Close SMTP server Task
    SMTP_close( hSMTP );
}
```

The above code assumes that the network scheduler Task can be launched whether or not the stack has a local IP address. This is true for servers that listen on a wildcard address of 0.0.0.0. In some rare cases, an IP address may be required for Task initialization, or perhaps an IP address on a certain device type is required. In these circumstances, the `NetworkIPAddr()` callback function signals the application that it is safe to start.

If you are using XGCONF for configuration, saving and reloading configurations via the *CfgSave()* and *CfgLoad()* functions is not automatically supported by XGCONF. However, internally, the same configuration database used by the Cfg* C functions is populated when the *.cfg file is built. You may want to use the example functions in Section 3.2.4.4 as hook functions to save the configuration created with XGCONF and reload if from non-volatile memory on startup.

3.1.5 Global Debug Configuration

There are two ways the stack can be shut down. The first is a manual shutdown that occurs when an application calls *NC_NetStop()*. The calling argument to the function is returned to the NETCTRL thread as the return value from *NC_NetStart()*. Therefore, for the example code, calling *NC_NetStop(1)* reboots the network stack, while calling *NC_NetStop(0)* shuts down the network stack.

The second way the stack can be shut down is when the stack code detects a debug message above the level you have set for shutdown control. You can configure this level by selecting the NDK's **Global** module and then clicking the **Debug** button.

The **Debug Print Message Level** controls which messages are sent to the debug log. For example, if you set this level to "Warning Messages", then warnings and errors will go to the debug log, but informational errors will not. By default, all messages are sent to the debug log.

The **Debug Abort Message Level** controls what types of messages trigger a stack shutdown. For example, if you set this level to "No Messages", then the stack is never shut down in response to an error. In this case, your application must detect and respond to messages. By default, only error messages trigger a stack shutdown.

3.1.6 Advanced Global Configuration

You can configure additional global NDK properties by selecting the NDK's **Global** module and then clicking the **Advance** button. You should be careful when setting these properties. In general, it is best to leave these properties set to their defaults. Some example advanced properties are:

- *Global.ndkTickPeriod* lets you adjust the NDK heartbeat rate. The default is 100 ticks. This matches the default SYS/BIOS Timer object, which drives the SYS/BIOS Clock and is configured so that 1 tick = 1 millisecond. However, you can configure a new Timer and use that to drive the Clock module. If that new Timer is not configured such that 1 tick = 1 millisecond, then you should also adjust the NDK tick period accordingly.
- *Global.ndkThreadPri* and *Global.ndkThreadStackSize* let you control the priority and stack size of the main NDK scheduler thread.
- *Global.netSchedulerOpMode* is set to either Polling Mode (NC_OPMODE_POLLING) or Interrupt Mode (NC_OPMODE_INTERRUPT), and determines when the scheduler attempts to execute. Interrupt mode is used in the vast majority of applications. Note that polling mode attempts to run continuously, so when polling is used, a low priority must be used.
- *Global.multiCoreStackRunMode* lets you control which cores (on a C6000 multi-core processor) run the NDK stack. By default, only core 0 runs the NDK stack. Set this property only if you are an advanced user of the Inter-Processor Communication (IPC) component.
- *Global.enableCodeGeneration* is set to true by default. If you set it to false, no C code is generated by the configuration, but the configuration still controls which NDK libraries are linked into the application.

3.1.7 Adding Clients and Servers

You can easily add support for additional modules to your application by enabling them in the configuration. For example, the following steps configure a static IP address:

1. Click on the IP module in the **System Overview** diagram or in the **Available Products** view.
2. In the **IP Settings: General Settings** page, check the box to **Add the IP to my configuration**.
3. Uncheck the box to **Obtain IP Address Automatically** to enable setting a static IP address.
4. Make settings similar to the following in this sheet.

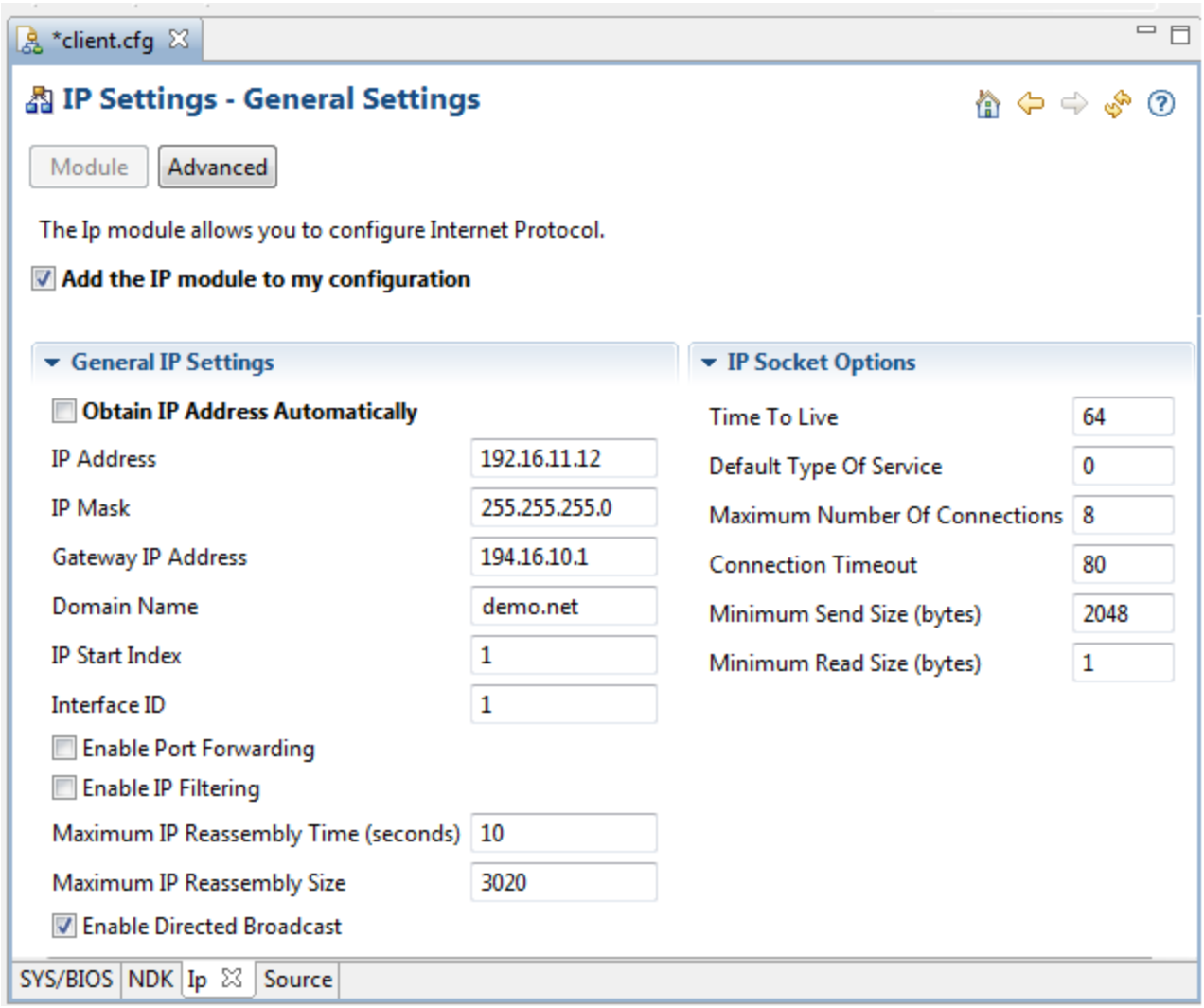


Figure 3-1. Configuring the IP Module

5. If you want information about a property, point to the field with your mouse cursor. Right-click on any field to get reference help for all the configurable IP module properties.
6. In addition to the properties listed on the General Settings page, a number of additional properties can be set if you click the **Advanced** button.

3.2 Configuring the NDK with C Code (without XGCONF)

If you are *not* using XGCONF for configuration, you must be aware of the additional development requirements described in this section.

If you use XGCONF to configure the NDK, you can ignore the following subsections.

3.2.1 Required SYS/BIOS Objects

The NDK is bolted to SYS/BIOS and the hardware via the OS adaptation layer and the HAL layer. These libraries do require SYS/BIOS objects to be created in order for them to work properly. This requirement can be altered by altering the OS and HAL layers.

The timer driver in the HAL requires that a SYS/BIOS Timer object be created to drive its main timer. The Timer must be configured to fire every 100mS, and call the timer driver function *llTimerTick()*.

The Task adaptation module in the OS library requires a hook to be able to save and load private environment pointers for the NDK. This is done by creating a SYS/BIOS hook. A hook module must be created to call the OS hook functions *NDK_hookInit()* and *NDK_hookCreate()*.

If you use XGCONF to configure the NDK, these objects are all created automatically.

3.2.2 Include Files

If you use XGCONF to configure the NDK, the correct include file directory is automatically referenced by your CCS project.

If you are using the legacy Cfg*() functions to add settings to the configuration database, you are responsible for pointing to the correct include file directory. The include directory in the NDK installation is described in Section 1.5.2. If you are not using XGCONF, you should include the base NDK include directory in the project build options of the CCStudio project. For example, with the default installation, the project should be set to include the include file path `<NDK_INSTALL_DIR>\packages\ti\ndk\inc`.

3.2.3 Library Files

If you use XGCONF to configure the NDK, the correct libraries are linked with the application automatically.

If you are using the legacy Cfg*() functions to add settings to the configuration database, you are responsible for linking the correct libraries into your project. If you are not using XGCONF, it is easiest to add the desired library files directly into the CCS project. This way, the linker will know where to find them.

3.2.4 System Configuration

If you are *not* using XGCONF, you must create a system configuration in order to be able to use the NETCTRL API. The configuration is a handle-based object that holds a multitude of system parameters. These parameters control the operation of the stack. Typical configuration parameters include:

- Network Hostname
- IP Address and Subnet Mask
- IP Address of Default Routes
- Services to be Executed (DHCP, DNS, HTTP, etc.)
- IP Address of name servers
- Stack Properties (IP routing, socket buffer size, ARP timeouts, etc.)

The process of creating a configuration always starts out with a call to *CfgNew()* to create a configuration handle. Once the configuration handle is created, configuration information can be loaded into the handle in bulk or constructed into it one entry at a time.

Loading a configuration in bulk requires that a previously constructed configuration has been saved to non-volatile storage. Once the configuration is in memory, the information can be loaded into the configuration handle by calling *CfgLoad()*. Another option is to manually add individual items to the configuration for the various desired properties. This is done by calling *CfgAddEntry()* for each individual entry to add.

The exact specification of the stack's configuration API appears in the Initialization and Configuration section of the *TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524). Some additional programming examples are provided in the Section 3.2.4.1 section of this document, and in the NDK example programs.

3.2.4.1 Configuration Examples

This section contains some sample code for constructing configurations without the use of XGCONF.

3.2.4.1.1 Constructing a Configuration for a Static IP and Gateway

The *NetworkTest()* function in this example consists of the main initialization thread for the stack. It creates a new configuration, adds a static IP address, subnet, and default gateway, and then boots up the stack.

In this case, it is assumed that the addressing and name information is stored in non-volatile memory. Here, we have defined some strings to hold the information. For example:

```
char *LocalIPAddr = "194.16.11.12";
char *LocalIPMask = "255.255.255.0";
char *GatewayIP = "194.16.10.1";
char *HostName = "testhost";
char *DomainName = "demo.net";
```

The code below performs the following operations :

1. Call *NC_SystemOpen()* and Create a new configuration.
2. Create and add a configuration entry for the local IP address and subnet using the supplied LocalIPAddr, LocalIPMask, and DomainName strings.
3. Create and add a configuration entry for the local hostname using the Hostname string.
4. Create and add a default route to the router supplied in the GatewayIP string.
5. Boot the system using this configuration by calling *NC_NetStart()*.
6. Free the configuration on system shutdown (when *NC_NetStart()* returns) and call *NC_SystemClose()*.

```
int NetworkTest()
{
    int rc;
    CI_IPNET NA;
    CI_ROUTE RT;
    HANDLE hCfg;

    //
    // THIS MUST BE THE ABSOLUTE FIRST THING DONE IN AN APPLICATION!!
    //
    rc = NC_SystemOpen( NC_PRIORITY_LOW, NC_OPMODE_INTERRUPT );
    if( rc )
    {
        printf("NC_SystemOpen Failed (%d)\n",rc);
        for(;;);
    }

    //
    // Create and build the system configuration from scratch.
    //

    // Create a new configuration
    hCfg = CfgNew();
    if( !hCfg )
    {
        printf("Unable to create configuration\n");
        goto main_exit;
    }

    // We'd better validate the length of the supplied names
    if( strlen( DomainName ) >= CFG_DOMAIN_MAX ||
        strlen( HostName ) >= CFG_HOSTNAME_MAX )
    {
        printf("Names too long\n");
        goto main_exit;
    }
```

```
    // Manually configure our local IP address
    bzero( &NA, sizeof(NA) );
    NA.IPAddr = inet_addr(LocalIPAddr);
    NA.IPMask = inet_addr(LocalIPMask);
    strcpy( NA.Domain, DomainName );
    NA.NetType = 0;

    // Add the address to interface 1
    CfgAddEntry( hCfg, CFGTAG_IPNET, 1, 0,
        sizeof(CI_IPNET), (UINT8 *)&NA, 0 );

    // Add our hostname
    CfgAddEntry( hCfg, CFGTAG_SYSINFO, CFGITEM_DHCP_HOSTNAME, 0,
        strlen(HostName), (UINT8 *)HostName, 0 );

    // Add the default gateway. Since it is the default, the
    // destination address and mask are both zero (we go ahead
    // and show the assignment for clarity).
    bzero( &RT, sizeof(RT) );
    RT.IPDestAddr = 0;
    RT.IPDestMask = 0;
    RT.IPGateAddr = inet_addr(GatewayIP);

    // Add the route
    CfgAddEntry( hCfg, CFGTAG_ROUTE, 0, 0, sizeof(CI_ROUTE), (UINT8 *)&RT, 0 );

    //
    // Boot the system using this configuration
    //
    // We keep booting until the function returns less than 1. This allows
    // us to have a "reboot" command.
    //
    do
    {
        rc = NC_NetStart( hCfg, NetworkStart, NetworkStop, NetworkIPAddr );
    } while( rc > 0 );

    // Delete Configuration
    CfgFree( hCfg );

// Close the OS
main_exit:
    NC_SystemClose();
    return(0);
}
```

3.2.4.1.2 Constructing a Configuration using the DHCP Client Service

In this section we take the initialization example of the previous section and alter it to instruct the stack to use the DHCP (Dynamic Host Configuration Protocol) client service to perform its IP address configuration.

Since DHCP provides the IP address, route, domain, and domain name servers, you only need to provide the hostname. The NetworkTest() function would look as follows (see the *TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524) for more details on using DHCP).

The code below performs the following operations :

1. Call *NC_SystemOpen()* and create a new configuration.
2. Create and add a configuration entry specifying the DHCP client service to be used.
3. Create and add a configuration entry for the local hostname using the Hostname string.
4. Boot the system using this configuration by calling *NC_NetStart()*.
5. Free the configuration on system shutdown (when *NC_NetStart()* returns) and call *NC_SystemClose()*.

```
char *HostName = "testhost";
```

```
int NetworkTest()
{
    int rc;
    CI_SERVICE_DHCPC dhcpc;
    HANDLE hCfg;

    //
    // THIS MUST BE THE ABSOLUTE FIRST THING DONE IN AN APPLICATION!!
    //
    rc = NC_SystemOpen( NC_PRIORITY_LOW, NC_OPMODE_INTERRUPT );
    if( rc )
    {
        printf("NC_SystemOpen Failed (%d)\n",rc);
        for(;;);
    }

    //
    // Create and build the system configuration from scratch.
    //

    // Create a new configuration hCfg = CfgNew();
    if( !hCfg )
    {
        printf("Unable to create configuration\n");
        goto main_exit; }

    // We'd better validate the length of the supplied names
    if( strlen( HostName ) >= CFG_HOSTNAME_MAX )
    {
        printf("Names too long\n");
        goto main_exit; }

    // Specify DHCP Service on interface 1
    bzero( &dhcpc, sizeof(dhcpc) );
    dhcpc.cisargs.Mode = CIS_FLG_IFIDXVALID;
    dhcpc.cisargs.IfIdx = 1;
    dhcpc.cisargs.pCbSrv = &ServiceReport;
    CfgAddEntry( hCfg, CFGTAG_SERVICE, CFGITEM_SERVICE_DHCPCLIENT, 0,
        sizeof(dhcpc), (UINT8 *)&dhcpc, 0 );

    // Add our hostname
    CfgAddEntry( hCfg, CFGTAG_SYSINFO, CFGITEM_DHCP_HOSTNAME, 0,
        strlen(HostName), (UINT8 *)HostName, 0 );

    //
    // Boot the system using this configuration
    //
    // We keep booting until the function returns less than 1. This allows
    // us to have a "reboot" command.
    //
    do
    {
        rc = NC_NetStart( hCfg, NetworkStart, NetworkStop, NetworkIPAddr );
    } while( rc > 0 );

    // Delete Configuration
    CfgFree( hCfg );

// Close the OS
main_exit:
    NC_SystemClose();
    return(0);
}
```

3.2.4.1.3 Using a Statically Defined DNS Server

The area of the configuration system that is used by the DHCP client can be difficult. When the DHCP client is in use, it has full control over the first 256 entries in the system information portion of the configuration system. In some rare instances, it may be useful to share this space with DHCP.

For example, assume a network application needs to manually add the IP address of a Domain Name System (DNS) server to the system configuration. When DHCP is not being used, this code is simple. To add a DNS server of 128.114.12.2, the following code would be added to the configuration build process (before calling *NC_NetStart()*).

```
IPN IPTmp;

// Manually add the DNS server "128.114.12.2"
IPTmp = inet_addr("128.114.12.2");

CfgAddEntry( hCfg, CFGTAG_SYSINFO, CFGITEM_DHCP_DOMAINNAMESERVER,
             0, sizeof(IPTmp), (UINT8 *)&IPTmp, 0 );
```

Note that the CLIENT example program in the example applications uses a form of this code. Now, when a DHCP client is used, it clears and resets the contents of the part of the configuration it controls. This includes the DNS server addresses. Therefore, if the above code was added to an application that used DHCP, the entry would be cleared whenever DHCP executed a status update.

To share this configuration space with DHCP (or to read the results of a DHCP configuration), the DHCP status callback report codes must be used. The status callback function was introduced in Section 3.2.5.5. When DHCP reports a status change, the application knows that the DHCP portion of the system configuration has been reset.

The following code also appears in the CLIENT example program. This code manually adds a DNS server address when the DHCP client is in use. Note that this code is part of the standard service callback function that is supplied to the configuration when the DHCP client service is specified.

```
//
// Service Status Reports
//
static char *TaskName[] = { "Telnet","HTTP","NAT","DHCPS","DHCPC","DNS" };
static char *ReportStr[] = { "","Running","Updated","Complete","Fault" };
static char *StatusStr[] = { "Disabled","Waiting","IPTerm", "Failed","Enabled" };

static void ServiceReport( uint Item, uint Status, uint Report, HANDLE h )
{
    printf( "Service Status: %-9s: %-9s: %-9s: %03d\n",
        TaskName[Item-1], StatusStr[Status], ReportStr[Report/256], Report&0xFF );

    // Example of adding to the DHCP configuration space
    //
    // When using the DHCP client, the client has full control over access
    // to the first 256 entries in the CFGTAG_SYSINFO space. Here, we want
    // to manually add a DNS server to the configuration, but we can only
    // do it once DHCP has finished its programming.
    //
    if( Item == CFGITEM_SERVICE_DHCPCLIENT &&
        Status == CIS_SRV_STATUS_ENABLED &&
        (Report == (NETTOOLS_STAT_RUNNING|DHCPCODE_IPADD) ||
        Report == (NETTOOLS_STAT_RUNNING|DHCPCODE_IPRENEW)) )
    {
        IPN IPTmp;

        // Manually add the DNS server when specified. If the address
        // string reads "0.0.0.0", IPTmp will be set to zero.
        IPTmp = inet_addr(DNSServer);

        if( IPTmp )
            CfgAddEntry( 0, CFGTAG_SYSINFO, CFGITEM_DHCP_DOMAINNAMESERVER,
                0, sizeof(IPTmp), (UINT8 *)&IPTmp, 0 );
    }
}
```

3.2.4.2 Controlling NDK and OS Options via the Configuration

Along with specifying IP addresses, routes, and services, the configuration system allows you to directly manipulate the configuration structures of the OS adaptation layer and the NDK. The OS configuration structure is discussed in the Operating System Configuration section of the *TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524), and the NDK configuration structure is discussed in the Configuring the Stack section in the appendices. The configuration interface to these internal structures is consolidated into a single configuration API as specified in the Initialization and Configuration section.

Although the values in these two configuration structures can be modified directly, adding the parameters to the system configuration is useful for two reasons. First, it provides a consistent API for all network configuration, and second, if the configuration load and save feature is used, these configuration parameters are saved along with the rest of the system configuration.

As a quick example of setting an OS configuration option, the following code makes a change to the debug reporting mechanism. By default, all debug messages generated by the NDK are output to the CCStudio output window. However, the OS configuration can be adjusted to print only messages of a higher severity level, or to disable the debug messages entirely.

Most of the example applications included with the NDK will raise the threshold of printing debug messages from the INFO level to the WARNING level. Here is how it appears in the source code:

```
 // We do not want to see debug messages less than WARNINGS
rc = DBG_WARN;

CfgAddEntry( hCfg, CFGTAG_OS, CFGITEM_OS_DBGPRINTLEVEL,
             CFG_ADDMODE_UNIQUE, sizeof(uint), (UINT8 *)&rc, 0 );
```

3.2.4.3 Shutdown

There are two ways the stack can be shut down. The first is a manual shutdown that occurs when an application calls *NC_NetStop()*. Here, the calling argument to the function is returned to the NETCTRL thread as the return value from *NC_NetStart()*. Therefore, for the example code, calling *NC_NetStop(1)* reboots the network stack, while calling *NC_NetStop(0)* shuts down the network stack.

The second way the stack can be shut down is when the stack code detects a fatal error. A fatal error is an error above the fatal threshold set in the configuration. This type of error generally indicates that it is not safe for the stack to continue. When this occurs, the stack code calls *NC_NetStop(-1)*. It is then up to you to determine what should be done next. The way the *NC_NetStart()* loop is coded determines if the system will shut down (as in the example), or simply reboot.

Note that the critical threshold to shut down can also be disabled. The following code can be added to the configuration to disable error-related shutdowns:

```
// We do not want the stack to abort on any error
uint rc = DBG_NONE;

CfgAddEntry( hCfg, CFGTAG_OS, CFGITEM_OS_DBGABORTLEVEL,
    CFG_ADDMODE_UNIQUE, sizeof(uint), (UINT8 *)&rc, 0 );
```

3.2.4.4 Saving and Loading a Configuration

Once a configuration is constructed, the application may save it off into non-volatile RAM so that it can be reloaded on the next cold boot. This is especially useful in an embedded system where the configuration can be modified at runtime using a serial cable, Telnet, or an HTTP browser.

If you are using XGCONF for configuration, saving and reloading configurations is not automatically supported by XGCONF. However, internally, the same configuration database used by the Cfg* C functions is populated when the *.cfg file is built. You may want to use the functions in the following subsections as hook functions to save the configuration created with XGCONF and reload if from non-volatile memory on startup.

3.2.4.4.1 Saving the Configuration

To save the configuration, convert it to a linear buffer, and then save the linear buffer off to storage. Here is a quick example of a configuration save operation. Note the `MyMemorySave()` function is assumed to save off the linear buffer into non-volatile storage.

```
int SaveConfig( HANDLE hCfg )
{
    UINT8 *pBuf;
    int size;

    // Get the required size to save the configuration
    CfgSave( hCfg, &size, 0 );

    if( size && (pBuf = malloc(size) ) )
    {
         CfgSave( hCfg, &size, pBuf );
         MyMemorySave( pBuf, size );
         Free( pBuf );
         return(1);
    }
    return(0);
}
```

3.2.4.4.2 Loading the Configuration

Once a configuration is saved, it can be loaded from non-volatile memory on startup. For this final NetworkTest() example, assume that another Task has created, edited, or saved a valid configuration to some storage medium on a previous execution. In this network initialization routine, all that is required is to load the configuration from storage and boot the NDK using the current configuration.

For this example, assume that the function `MyMemorySize()` returns the size of the configuration in a stored linear buffer and that `MyMemoryLoad()`loads the linear buffer from non-volatile storage.

```
int NetworkTest()
{
    int rc;
    HANDLE hCfg;
    UINT8 *pBuf;
    Int size;

    //
    // THIS MUST BE THE ABSOLUTE FIRST THING DONE IN AN APPLICATION!!
    //
    rc = NC_SystemOpen( NC_PRIORITY_LOW, NC_OPMODE_INTERRUPT );
    if( rc )
    {
        printf("NC_SystemOpen Failed (%d)\n",rc);
        for(;;);
    }

    //
    // First load the linear memory block holding the configuration
    //

    // Allocate a buffer to hold the information
    size = MyMemorySize();
    if( !size )
        goto main_exit;

    pBuf = malloc( size );
    if( !pBuf )
        goto main_exit;

    // Load from non-volatile storage
    MyMemoryLoad( pBuf, size );
```

```
    //
    // Now create the configuration and load it
    //

    // Create a new configuration
    hCfg = CfgNew();

    if( !hCfg )
    {
        printf("Unable to create configuration\n");
        free( pBuf );
        goto main_exit;
    }

    // Load the configuration (and then we can free the buffer)
    CfgLoad( hCfg, size, pBuf );

    mmFree( pBuf );

    //
    // Boot the system using this configuration
    //
    // We keep booting until the function returns less than 1. This allows
    // us to have a "reboot" command.
    //
    do
    {
        rc = NC_NetStart( hCfg, NetworkStart, NetworkStop, NetworkIPAddr );
    } while( rc > 0 );

    // Delete Configuration
    CfgFree( hCfg );

// Close the OS
main_exit:
    NC_SystemClose();
    return(0);
}
```

3.2.5 NDK Initialization

Before a sockets application like the example shown in Section 3.4 can be executed, the stack must be properly configured and initialized. To facilitate a standard initialization process, and yet allow customization, source code to the network control module (NETCTRL) is included in the NDK. The NETCTRL module is the center of the stack's initialization and event scheduling. A solid comprehension of NETCTRL's operation is essential for building a solid networking application. This section describes how to use NETCTRL in an networking application. An explanation of how NETCTRL works and how it can be tuned is provided in Chapter 4.

The process of initialization of the NDK is described in detail in Chapter 4 of the *TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524). This section closely mirrors the initialization procedure described in the NDK Software Directory of that document. Here we describe the information with a more practical slant. Programmers concerned with the exact API of the functions mentioned here should refer to the *TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524) for a more precise description.

3.2.5.1 The NETCTRL Task Thread

If you are not using XGCONF, your application must create a Task thread that contains a call to *ND_NetStart()*, which in turn runs the network scheduler function. The NSP example applications provide this thread.

This Task thread (called the scheduler thread) is the thread in which nearly all the NETCTRL activity takes place. This thread acts as the program's entry-point and performs initialization actions. Later, it becomes the NETCTRL scheduler thread. Therefore, control of this thread is not returned to the caller until the stack has been shut down. Application Tasks—network-oriented or otherwise—are not executed within this thread.

3.2.5.2 Pre-Initialization

If you are not using XGCONF, your application must call the primary initialization function *NC_SystemOpen()* before calling any other of the stack API functions. This initializes the stack and the memory environment used by all the stack components. Two calling arguments, *Priority* and *OpMode*, indicate how the scheduler should execute. For example, the example applications included in the NSP contain the following code:

```
//
// THIS IS THE FIRST THING DONE IN AN APPLICATION!!
//
rc = NC_SystemOpen( NC_PRIORITY_LOW, NC_OPMODE_INTERRUPT );
if( rc )
{
    printf("NC_SystemOpen Failed (%d)\n",rc);
    for(;;);
}
```

3.2.5.3 Invoking New Network Tasks and Services

Some standard network services can be specified in the NDK configuration; these are loaded and unloaded automatically by the NETCTRL module. Other services, including those written by an applications programmer should be launched from callback functions.

If you are *not* using XGCONF, you can use the Start callback function supplied to *NC_NetStart()* to add a callback function. As an example of a network start callback, the NetworkStart() function below opens a user SMTP server application by calling an open function to create the main application thread.

```
static SMTP_Handle hSMTP;


//
// NetworkStart
//
// This function is called after the configuration has booted
//
static void NetworkStart( )
{
    // Create an SMTP server Task
    hSMTP = SMTP_open( );
}
```

The above code launches a self contained application that needs no further monitoring, but the application must be shut down when the system shuts down. This is done via the NetworkStop() callback function. Therefore, the NetworkStop() function must undo what was done in NetworkStart().

```
//
// NetworkStop
//
// This function is called when the network is shutting down
//
static void NetworkStop()
{
    // Close our SMTP server Task
    SMTP_close( hSMTP );
}
```

The above example assumes that the network scheduler Task can be launched whether or not the stack has a local IP address. This is true for servers that listen on a wildcard address of 0.0.0.0. In some rare cases, an IP address may be required for Task initialization, or perhaps an IP address on a certain device type is required. In these circumstances, the `NetworkIPAddr()` callback function signals the application that it is safe to start.

The following example illustrates the calling parameters to the `NetworkIPAddr()` callback. Note that the *IFIndexGetHandle()* and *IFGetType()* functions can be called to get the type of device (HTYPE_ETH or HTYPE_PPP) on which the new IP address is being added or removed. This example just prints a message. The most common use of this callback function is to synchronize network Tasks that require a local IP address to be installed before executing.

```
//
// NetworkIPAddr
// This function is called whenever an IP address binding is
// added or removed from the system.
//
static void NetworkIPAddr( IPN IPAddr, uint IfIdx, uint fAdd )
{
    IPN IPTmp;

    if( fAdd )
        printf("Network Added: ");
    else
        printf("Network Removed: ");

    // Print a message
    IPTmp = ntohl( IPAddr );

    printf("If-%d:%d.%d.%d.%d\n", IfIdx, (UINT8)(IPTmp>>24)&0xFF,
        (UINT8)(IPTmp>>16)&0xFF, (UINT8)(IPTmp>>8)&0xFF, (UINT8)IPTmp&0xFF );
}
```

3.2.5.4 Network Startup

If you are *not* using XGCONF, your application must call the NETCTRL function *NC_NetStart()* to invoke the network scheduler after the configuration is loaded. Besides the handle to the configuration, this function takes three additional callback pointer parameters; a pointer to a Start callback function, a Stop function, and a IP Address Event function.

The first two callback functions are called only once. The Start callback is called when the system is initialized and ready to execute network applications (note there may not be a local IP network address installed yet). The Stop callback is called when the system is shutting down and signifies that the stack will soon not be able to execute network applications. The third callback can be called multiple times. It is called when a local IP address is either added or removed from the system. This can be useful in detecting new DHCP or PPP address events, or just to record the local IP address for use by local network applications. The call to *NC_NetStart()* will not return until the system has shut down, and then it returns a shutdown code as its return value. How the system was shut down may be important to determine if the stack should be rebooted. For example, a reboot may be desired in order to load a new configuration. The return code from *NC_NetStart()* can be used to determine if *NC_NetStart()* should be called again (and hence perform the reboot).

For a simple example, the following code continuously reboots the stack using the current configuration handle if the stack shuts down with a return code greater than zero. The return code is set when the stack is shutdown via a call to *NC_NetStop()*.

```
//
// Boot the system using our configuration
//
// We keep booting until the function returns 0. This allows
// us to have a "reboot" command.
//
do
{
    rc = NC_NetStart( hCfg, NetworkStart, NetworkStop, NetworkIPAddr );
} while( rc > 0 );
```

3.2.5.5 Adding Status Report Services

The configuration system can also be used to invoke the standard network services found in the NETTOOLS library. The services available to network applications using the NDK are discussed in detail in Chapter 4 of the *TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524). This section summarized the services described in that chapter.

When using the NETTOOLS library, the NETTOOLS status callback function is introduced. This callback function tracks the state of services that are enabled through the configuration. There are two levels to the status callback function. The first callback is made by the NETTOOLS service. It calls the configuration service provider when the status of the service changes. The configuration service provider then adds its own status to the information and calls back to the application's callback function. A pointer to the application's callback is provided when the application adds the service to the system configuration.

If you are *not* using XGCONF, the basic status callback function that is used in all the examples is as follows:

```
//
// Service Status Reports
//
static char *TaskName[] = { "Telnet","HTTP","NAT","DHCPS","DHCPC","DNS" };
static char *ReportStr[] = { "","Running","Updated","Complete","Fault" };
static char *StatusStr[] = { "Disabled", "Waiting", "IPTerm", "Failed", "Enabled" }

static void ServiceReport( uint Item, uint Status, uint Report, HANDLE h )
{
    printf( "Service Status: %-9s: %-9s: %-9s: %03d\n",
        TaskName[Item-1], StatusStr[Status], ReportStr[Report/256], Report&0xFF );
}
```

Note that the names of the individual services are listed in the *TaskName[]* array. This order is specified by the definition of the service items in the configuration system and is constant. See the file INC\NETTOOLS\NETCFG.H for the physical declarations.

Note that the strings defining the master report code are listed in the `ReportStr[]` array. This order is specified by the NETTOOLS standard reporting mechanism and is constant. See the file INC\NETTOOLS\NETTOOLS.H for the physical declarations.

Note that the strings defining the Task state are defined in the `StatusStr[]` array. This order is specified by the definition of the standard service structure in the configuration system. See the file INC\NETTOOLS\NETCFG.H for the physical declarations.

The last value this callback function prints is the least significant 8 bits of the value passed in *Report*. This value is specific to the service in question. For most services this value is redundant. Usually, if the service succeeds, it reports Complete, and if the service fails, it reports Fault. For services that never complete (for example, a DHCP client that continues to run while the IP lease is active), the upper byte of *Report* signifies Running and the service specific lower byte must be used to determine the current state.

For example, the status codes returned in the 8 least significant bits of Report when using the DHCP client service are:

```
DHCPCODE_IPADD       Client has added an IP address
DHCPCODE_IPREMOVE    IP address removed and CFG erased
DHCPCODE_IPRENEW     IP renewed, DHCP config space reset
```

These DHCP client specific report codes are defined in INC\NETTOOLS\INC\DHCPIF.H. In most cases, you do not have to examine state report codes down to this level of detail, except in the following case. When using the DHCP client to configure the stack, the DHCP client controls the first 256 entries of the CFGTAG_SYSINFO tag space. These entries correspond to the 256 DHCP option tags. An application may check for DHCPCODE_IPADD or DHCPCODE_IPRENEW return codes so that it can read or alter information obtained by DHCP client. This is discussed further in Section 3.2.4.1.2.

3.3 Creating a Task

The process of creating a sockets application begins with the creation of a SYS/BIOS Task thread. You can use XGCONF to statically configure Tasks or use the standard SYS/BIOS API or the provided Task abstraction to dynamically create Tasks. For example, the following C code creates a basic Task:

```
Task_Params taskParams;
Task_Handle hMyTask;
Error_Block eb;

Error_init(&eb);

/* Create a Task with priority 5 */
Task_Params_init(&taskParams);
taskParams.stackSize = 4096;
taskParams.priority = 5;
hMyTask = Task_create((Task_FuncPtr)entrypoint, &taskParams, &eb);
if (hMyTask == NULL) {
    System_abort("Task create failed");
}
```

The same Task can be created via the *TaskCreate()* function in the Task abstraction API. The abstracted function is a little more restrictive. It creates a Task thread with exactly 3 parameters (they do not all have to be used). For example, the following call would create a Task similar to that shown above:

```
hMyTask = TaskCreate( entrypoint, "TaskName", OS_TASKPRINORM, stacksize, arg1, arg2, arg3 );
```

In both cases, hMyTask is a handle to a SYS/BIOS Task thread.

3.3.1 Initializing the File Descriptor Table

Each Task thread that must use the sockets or file API included in the stack must allocate a file descriptor table and associate the table with the Task handle. This process is described fully in the *TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524). Basically, a call to *fdOpenSession()* must be performed before any file descriptor oriented functions are used, and then *fdCloseSession()* is called when they are no longer required.

3.4 Example Code

The following is an echo sockets application for SYS/BIOS. It is adapted from code in the <NDK_INSTALL_DIR>\packages\ti\ndk\tools\console\conecho.c file. This code creates a socket, connects to port 7, sends some data, and then tries to receive it back.

The lines of code in **boldface** represent new functions required to provide sockets functionality to SYS/BIOS. The functions in ***bold italics*** are standard, but their names have been adjusted to avoid naming conflicts with the runtime support library provided by TI's code generation tools. The remainder of the functions should be familiar to Berkeley sockets programmers. All of these functions are described in detail in the *TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524).

```
void EchoTcp( IPN IPAddr )
{
    SOCKET s = INVALID_SOCKET;
    struct sockaddr_in sin1;
    int I;
    char *pBuf = 0;
    struct timeval timeout;

    // Allocate the file descriptor environment for this Task
    fdOpenSession( (HANDLE)Task_self() );

    printf("\n== Start TCP Echo Client Test ==\n");

    // Create test socket
    s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if( s == INVALID_SOCKET )
    {
        printf("failed socket create (%d)\n",fdError());
        goto leave;
```

```
    }
    // Prepare address for connect
    bzero( &sin1, sizeof(struct sockaddr_in) );
    sin1.sin_family = AF_INET;
    sin1.sin_len = sizeof( sin1 );
    sin1.sin_addr.s_addr = IPAddr;
    sin1.sin_port = htons(7);

    // Configure our Tx and Rx timeout to be 5 seconds
    timeout.tv_sec = 5;
    timeout.tv_usec = 0;
    setsockopt( s, SOL_SOCKET, SO_SNDTIMEO, &timeout, sizeof( timeout ) );
    setsockopt( s, SOL_SOCKET, SO_RCVTIMEO, &timeout, sizeof( timeout ) );
    // Connect socket
    if( connect( s, (PSA) &sin1, sizeof(sin1) ) < 0 )
    {
        printf("failed connect (%d)\n",fdError());
        goto leave;
    }

    // Allocate a working buffer
    if( !(pBuf = malloc( 4096 )) )
    {
        printf("failed temp buffer allocation\n");
        goto leave;
    }
    // Fill buffer with a test pattern
    for(I=0; i<4096; I++)
        *(pBuf+I) = (char)I;
    // Send the buffer
    if( send( s, pBuf, 4096, 0 ) < 0 )
    {
        printf("send failed (%d)\n",fdError());
        goto leave;
    }
    // Try and receive the test pattern back
    I = recv( s, pBuf, 4096, MSG_WAITALL );
    if( I < 0 )
    {
        printf("recv failed (%d)\n",fdError());
        goto leave;
    }

    // Verify reception size and pattern
    if( I != test )
    {
        printf("received %d (not %d) bytes\n",i, test);
        goto leave;
    }
    for(I=0; i<test; I++)
        if( *(pBuf+I) != (char)I )
        {
            printf("verify failed at byte %d\n",I);
            break;
        }

    // If here, the test passed
    if( I==test )
        printf("passed\n");
leave:
    if( pBuf )
        free( pBuf );

    if( s != INVALID_SOCKET )
```

```
        fdClose( s );
    printf("== End TCP Echo Client Test ==\n\n");

    // Free the file descriptor environment for this Task
    fdCloseSession( (HANDLE)Task_self() );
    Task_exit();
}
```

NOTE: The example code above only illustrates IPv4 sockets (AF_INET family). For sample illustration of IPv6 sockets, refer to example code provided in the *conipv6.c* file packaged as a part of the client example project for your platform.

3.5 Application Debug and Troubleshooting

Although there is certainly no instant or easy way to debug an NDK application, the following sections provide a quick description of some of the potential problem areas. Some of these topics are discussed elsewhere in the documentation as well.

3.5.1 Troubleshooting Common Problems

One of the most common support requests for the NDK deals with the inability to either send or receive network packets. This may also take the form of dropping packets or general poor performance. There are many causes for this type of behavior. For potential scheduling issues, see Section 3.1.2.2. It is also recommended that application programmers fully understand the workings of the NETCTRL module. For this, see Chapter 4.

Here is a quick list. If you are using XGCONF for configuration, many of the potential configuration problems cannot occur.

All socket calls return “error” (-1)

- Make sure there is a call to *fdOpenSession()* in the Task before it uses sockets, and a call to *fdCloseSession()* when the Task terminates.

No link indication, or will not re-link when cable is disconnected and reconnected.

- Make sure there is a Timer object in your SYS/BIOS configuration that is calling the driver function *llTimerTick()* every 100 ms.

Not receiving any packets – ever

- When polling for data by making *recv()*, *fdPoll()*, or *fdSelect()* calls in a non-blocking fashion, make sure you do not have any scheduling issues. When the NETCTRL scheduler is running in low priority, network applications are not allowed to poll without blocking. Try running the scheduler in high priority (via *NC_SystemOpen()*).
- The NDK assumes there is some L2 cache. If the DSP or ARM is configured to *all internal memory* with nothing left for L2 cache, the NDK drivers will not function properly.

Performance is sluggish. Very slow ping response.

- Make sure there is a Timer object in your SYS/BIOS configuration that is calling the driver function *llTimerTick()* every 100 ms.
- If porting an Ethernet driver and running NETCTRL in interrupt mode, make sure your device is correctly detecting interrupts. Make sure the interrupt polarity is correct.

UDP application drops packets on send() calls.

- If sending to a new IP address, the very first send may be held up in the ARP layer while the stack determines the MAC address for the packet destination. While in this mode, subsequent sends are discarded.
- When using UDP and sending multiple packets at once, make sure you have plenty of packet buffers available (see Section 5.3.1).
- Verify you do not have any scheduling issues. Try running the scheduler in high priority (via *NC_SystemOpen()*).

UDP application drops packets on recv() calls.

- Make sure you have plenty of packet buffers available (see Section 5.3.1).
- Make sure the packet threshold for UDP is high enough to hold all UDP data received in between calls to *recv()* (see CFGITEM_IP_SOCKUDPRXLIMIT in the *NDK Programmer's Reference Guide*).
- Verify you do not have any scheduling issues. Try running the scheduler in high priority (via *NC_SystemOpen()*).
- It is possible that packets are being dropped by the Ethernet device driver. Some device drivers have adjustable RX queue depths, while others do not. Refer to the source code of your Ethernet device driver for more details (device driver source code is provided in NDK Support Package for your hardware platform).

Pings to NDK target Fails Beyond 3012 Size

The NDK's default configuration allows reassembly of packets up to "3012" bytes. To be able to ping bigger sizes, the stack needs to be reconfigured as follows:

- Change the "MMALLOC_MAXSIZE" definition in "pbm.c" file. (i.e. #define MMALLOC_MAXSIZE 65500) and rebuild the library.
- Increase the Memory Manager Buffer **Page Size** in the **Buffers** tab of the Global configuration.
- Increase the **Maximum IP Reassembly Size** property of the IP module configuration.

Sending and Receiving UDP Datagrams over MTU Size

The size of sending and receiving UDP datagrams are dependent on the following NDK configuration options, socket options, and OSAL layer definitions:

- NDK Configuration Options:
  - Increase the **Minimum Send Size** property of the IP module socket configuration. *See the TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524).
  - Increase the **Minimum Read Size** property of the IP module socket configuration.
  - If you are *not* using XGCONF for configuration, you can configure these IP module properties by using the following C code:

```
uint tmp = 65500;

// configure NDK
CfgAddEntry(hCfg, CFGTAG_IP, CFGITEM_IP_IPREASMMAXSIZE,
            CFG_ADDMODE_UNIQUE, sizeof(uint), (UINT8*) &tmp, 0);
CfgAddEntry(hCfg, CFGTAG_IP, CFGITEM_IP_SOCKUDPRXLIMIT,
            CFG_ADDMODE_UNIQUE, sizeof(uint), (UINT8*) &tmp, 0);

// set socket options
setsockopt(s, SOL_SOCKET, SO_RCVBUF, &tmp, sizeof(int) );
setsockopt(s, SOL_SOCKET, SO_SNDBUF, &tmp, sizeof(int) );
```

- Socket Options:
  - SO_SNDBUF: *See the TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524)
  - SO_RCVBUF - *See the TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524)
- OSAL Layer Definitions:
  - Change the "MMALLOC_MAXSIZE" definition in "pbm.c" file. (i.e. #define MMALLOC_MAXSIZE 65500) and rebuild the library
  - Increase the Memory Manager Buffer **Page Size** in the **Buffers** tab of the Global configuration.
  - If you are *not* using XGCONF for configuration, you can edit the MMALLOC_MAXSIZE definition in the pbm.c file and RAW_PAGE_SIZE definition in the mem.c file. Then rebuild the appropriate OSAL library in <NDK_INSTALL_DIR>\packages\ti\ndk\os\lib.

Timestamping UDP Datagram Payloads

The NDK allows the application to update the payload of UDP datagrams. The typical usage of this is to update the timestamp information of the datagram. This way, transmitting and receiving ends can more accurately adjust delivery delays depending on changing run-time characteristic of the system.

On the transmitting end:

- The application can register a call-out function per socket basis by using the "setsockopt()" function.
- The call-out function is called by the stack before inserting the datagram into driver's transmit queue.
- It is the call-out function's responsibility to update the UDP checksum information in the header.
- The following code section is a sample of how to control it:

```
void myTxTimestampFxn(UINT8 *pIpHdr) {
    ...
}

setsockopt(s, SOL_SOCKET, SO_TXTIMESTAMP, (void*) myTxTimestampFxn, sizeof(void*));
```

On the receiving end:

- The application can register a call-out function per interface basis by using the "EtherConfig()" function. It is set in the "NC_NetStart()" function of "netctrl.c".
- The call-out function is called by the stack scheduler just before processing the packet.
- It is the call-out function's responsibility to update the UDP checksum information in the header.
- The following code section is a sample of how to control it:

```
void myRcvTimestampFxn(UINT8 *pIpHdr) {
    ...
}

EtherConfig( hEther[i], 1518, 14, 0, 6, 12, 4, myRcvTimestampFxn);
```

In General

- Do not try to tune the Timer function frequency. Make sure it calls *llTimerTick()* every 100 ms.
- Watch for out of memory conditions. These can be detected by the return from some functions, but will also print out warning messages when the messages are enabled. These messages contain the acronym OOM for out of memory. (Out of memory conditions can be caused by many things, but the most common cause in the NDK is when TCP sockets are created and closed very quickly without using the SO_LINGER socket option. This puts many sockets in the TCP timewait state, exhausting scratchpad memory. The solution is to use the SO_LINGER socket option.)

3.5.2 Controlling Debug Messages

Most of the text messages generated by a network application come from the application. However, it is possible for the network stack to generate debug messages.

The NDK includes its own debug message system. This system can be ported to behave in any manner desired, but by default, debug messages are printed to the debugger using an internal *printf()* function.

Debug messages also include an associated severity level. These levels are DBG_INFO, DBG_WARN, and DBG_ERROR. The severity level is used for two purposes. First, it determines whether or not the debug message will be printed, and second, it determines whether or not the debug message will cause the NDK to shutdown.

By default, all debug messages are printed, and messages with a level of DBG_ERROR causes a stack shutdown. This behavior can be modified by using XGCONF as described in Section 3.1.5. Or, you can modify it through the system configuration as described in Section 3.2.4.2 and Section 3.2.4.3. Also see the *TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524).

3.5.3 Interpreting Debug Messages

The following is a list of some of the debug messages that may occur during stack operation, along with the most commonly associated cause.

3.5.3.1 TCP: Retransmit Timeout: Level DBG_INFO

This message is generated by TCP when it has sent a packet of data to a network peer, and the peer has not replied in the expected amount of time. This can be just about anything; the peer has gone down, the network is busy, the network packet was dropped or corrupted, and so on.

3.5.3.2 FunctionName: Buffer OOM: Level DBG_WARN

This message is generated by some modules when unexpected out of memory conditions occur. The stack has an internal resource recovery routine to help deal with these situations; however, a significant number of these messages may also indicate that there is not enough large block memory available, or that there is a memory leak. See the notes on the memory manager reports in this section for more details.

3.5.3.3 mmFree: Double Free: Level DBG_WARN

A double free message occurs when the mmFree() function is called on a block of memory that was not marked as allocated. This can be caused by physically calling mmFree() twice for the same memory, but more commonly is caused by memory corruption. See Section 3.5.4 for possible causes.

3.5.3.4 FunctionName: HTYPE nnnn: Level DBG_ERROR

This message is generated only by the strong checking version of the stack. It is caused when a handle is passed to a function that is not of the proper handle type. Since the object oriented nature of the stack is hidden from the network applications writer, this error should never occur. If it is not caused by the attempt to call internal stack functions, then it is most likely the result of memory corruption. See the notes on memory corruption in this section for possible causes.

3.5.3.5 mmAlloc: PIT ???? Sync: Level DBG_ERROR

This message is generated by the scratch memory allocation system. PIT is an acronym for page information table. Table synchronization errors can only be caused by memory corruption. See Section 3.5.4 for possible causes.

3.5.3.6 PBM_enq: Invalid Packet: Level DBG_ERROR

This message is generated by the packet buffer manager (PBM) module driver in the OS adaptation layer. When the PBM module initially allocates its packet buffer pool, it marks each packet buffer with a magic number. During normal operation, packets are pushed and popped to and from various queues. On each push operation, the packet's magic number is checked. When the magic number is invalid, this message results. It is possible for an invalid packet to be introduced into the system when using the non copy sockets API extensions, but the vastly more common cause is memory corruption. See the notes on memory corruption in this section for possible causes.

3.5.4 Memory Corruption

The words memory corruption come up frequently when diagnosing NDK debug messages. This is because it is easy to corrupt memory on cache devices. Most of the example programs included in the NDK run using full L2 cache. In this mode, any read or write access to the internal memory range of the CPU can cause cache corruption and hence cause memory corruption. Since the internal memory range starts at address 0x00000000, a NULL pointer can cause problems when using full cache.

To check to see if corruption is being caused by a NULL pointer, change the cache mode to use less cache. When there is some internal memory available, reads or writes to address 0x0 do not cause cache corruption (the application still may not work, but the error messages should stop).

Another way to track down any kind of cache corruption is to break on CPU reads or writes to the entire cache range. Code Composer Studio has the ability to trap reads or writes to a range of memory, but both cannot be checked simultaneously. Therefore, a couple of trials may be necessary.

Of course, it is possible that the memory corruption has nothing to do with the stack. It could be a wild pointer. However, since corrupting the cache can corrupt memory throughout the system, the cache is the first place to start.

3.5.5 Program Lockups

Most lockup conditions are caused by insufficient Task stack sizes. For example, when writing an HTTP CGI function, the CGI function Task thread has only about 5000 bytes of total Task stack. Therefore, using large amounts of stack is not recommended. In general, do not use the following code:

```
myTask()
{
    char TempBuffer[2000];
    myFun( TempBuffer );
}
```

but instead, use the following:

```
myTask()
{
    char *pTempBuf;

    pTempBuf = Memory_alloc( NULL, 2000, 0, &eb )

    if (pTempBuf != NULL)
    {
        myFun( pTempBuf );
        Memory_free( NULL, pTempBuf, 2000 );
    }
}
```

If calling a memory allocation function is too much of a speed overhead, consider using an external buffer.

This is just an example, with a little forethought you can eliminate all possible stack overflow conditions, and eliminate the possibility of program lockups from this condition.

3.5.6 Memory Management Reports

The memory manager that manages scratch memory in the NDK has a built in reporting system. It tracks the use of scratch memory closely (calls to *mmAlloc()* and *mmFree()*), and also tracks calls to the large block memory allocated (calls to *mmBulkAlloc()* and *mmBulkFree()*). Note that the bulk allocation functions simply call *malloc()* and *free()*. This behavior can be altered by adjusting the memory manager.

The memory report is shown below. It lists the max number of blocks allocated per size bucket, the number of calls to malloc and free, and a list of allocated memory. An example report is shown below:

```
48:48   ( 75%)   18:96   ( 56%)   8:128  ( 33%)      28:256 ( 77%)
 1:512  ( 16%)    0:1536          0:3072
(21504/46080 mmAlloc: 61347036/0/61346947, mmBulk: 25/0/17)

1 blocks alloced in 512 byte page
38 blocks alloced in 48 byte page
18 blocks alloced in 96 byte page
8 blocks alloced in 128 byte page
12 blocks alloced in 256 byte page
12 blocks alloced in 256 byte page
```

Here, the entry 18:96 (56%) means that at most, 18 blocks were allocated in the 96 byte bucket. The page size on the memory manager is 3072, so 56% of a page was used. The entry 21504/46080 means that at most 21,504 bytes were allocated, with a total of 46,080 bytes available.

The entry mmAlloc: 61347036/0/61346947 means that 61,347,036 calls were made to *mmAlloc()*, of which 0 failed, and 61,346,947 calls were made to *mmFree()*. Note that at any time, the call to mmAlloc plus the failures must equal the calls to mmFree plus any outstanding allocations. Therefore, on a final report were the report is mmAlloc: n1/n2/n3, n1+n2 should equal n3. If not, there is a memory leak.

There are several methods to obtain a memory report when using the telnet console program included with most of the example applications. The console 'mem' command prints out a current report, but more importantly, the console 'shutdown' command shuts down the stack and prints out a final report. If all network applications are created and destroyed according to the specifications in this document, there should be no memory leaks detected in the final report. The function called to obtain a memory report is defined below.

3.5.6.1 mmCheck – Generate Memory Manager Report

mmCheck ***Generate Memory Manager Report***

Syntax

void _mmCheck(uint CallMode, int (*pPrn)(const char *,...));

Parameters

CallMode	Specifies the type of report to generate
pPrn	Pointer to printf() compatible function

Description

Prints out a memory report to the printf() compatible function pointed to by pPrn. The type of report printed is determined by the value of CallMode. The reporting function has the option of printing out memory block IDs. This means that the first uint sized field in the memory block of each allocated block is printed in the report. This is a useful option when the first field of allocated memory stores an object handle type, or some other unique identifier.

Call Mode

Can be set to one of the following:

MMCHECK_MAP	Map out allocated memory, but do not dump ID's
MMCHECK_DUMP	Dump allocated block IDs
MMCHECK_SHUTDOWN	Dump allocated block IDs & free scratchpad memory

Note: Do not attempt to use any mmAlloc() functions after requesting a MMCHECK_SHUTDOWN report!

Returns

None

Network Control Functions

This chapter describes the network control functions.

Topic **Page**

4.1 Introduction to NETCTRL Source

4.1.1 History

The NETCTRL module was originally a recommended initialization and scheduling method to execute the NDK. Although mostly simple, this code became standard. Eventually, it was separated out into the NETCTRL library.

The NETCTRL module is the center of the NDK because it connects the HAL and the OS adaptation layer to the NDK. It controls both initialization and how events are scheduled for execution within the stack. Understanding how the NETCTRL module works helps you tune your DSP or ARM networking application for ideal performance.

4.1.2 NETCTRL Source Files

Source code to the NETCTRL library consists of two C files located in the <NDK_INSTALL_DIR>\packages\ti\ndk\netctrl directory:

NETCTRL.C	Network Control (Initialization and Scheduling) Module
NETSRV.C	Configuration service module (system configuration service provider)

There are two include files associated with NETCTRL in the \INC\NETCTRL directory:

NETCTRL.H	Interface specification to NETCTRL
NETSRV.H	Interface specification to NETSRV

4.1.3 Main Functions

The NETCTRL.C source module contains source code for all the functions with the NC_ prefix. The function of the NETCTRL module has three basic parts.

The first function of NETCTRL.C is to perform the system initialization and shutdown that is necessary before calling any other stack functions. These functions are declared as *NC_SystemOpen()* and *NC_SystemClose()*.

The second function of NETCTRL.C is to perform the driver environment initialization and configuration bootstrap necessary to start the stack functionality. This startup function and its shutdown counterpart are declared as *NC_NetStart()* and *NC_NetStop()*.

The final function of NETCTRL.C that is hidden from the caller, is implementing the stack's event scheduling, which is the center of the stack's operation.

The NETSRV.C module contains the code that boots all the services on the stack. This code takes what is stored in the stack’s configuration and implements the necessary stack functions to keep the configuration current. When an active item in the configuration is changed, there is code in the NETSRV module to execute that change in the NDK.

4.1.4 Additional Functions

There are some additional NETCTRL functions that are not documented in the *TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524). These functions are *NC_BootComplete()* and *NC_IPUpdate()*. They are both called from the NETSRV module.

The *NC_NetStart()* function initiates the configuration boot process by creating a boot thread with an entry point of *NS_BootTask()* (from NETSRV.C). The *NC_BootComplete()* function is called by the configuration boot thread when the configuration boot is complete. It signals to NETCTRL that it can now call the *NetworkStart()* application callback that was passed to NC_NetStart() by the application. On return from *NC_BootComplete()*, the boot thread is terminated. Therefore, the application programmer may take control of the *NetworkStart()* callback thread, although this is not recommended.

The IP address update function is called by NETSRV when an address is added to or removed from the system. It is this function that then calls the NetworkIPAddr() application callback that was originally passed to *NC_NetStart()*.

4.1.5 Booting and Scheduling

Section 3.2.5 discussed using the network control (NETCTRL) module. This section examines the internal source code of the main NETCTRL module and the operation of the event scheduler.

The stack event scheduler is the routine that calls the stack to process packet and timer events. The scheduler is called from within *NC_NetStart()* and does not return until the stack is being shut down, which explains why the *NC_NetStart()* function does not return to the application until the system is shut down and the scheduler terminates.

The basic flow of *NC_NetStart()* is as follows:

```
NC_NetStart()
{
    Initialize_Devices();

    CreateConfigurationBootThread() ;
    NetScheduler();
    CloseConfiguration();
    CloseDevices();
}
```

Out of the functional stages for *NC_NetStart()* listed above, the two that are of the most concern are the creation of the boot thread, and the implementation of the network event scheduler.

The boot thread is handled by a second C module in the NETCTRL library named NETSRV.C. This name is an abbreviation for Network Service Manager. The NETSRV module hooks into the configuration system as a configuration service provider. The configuration system module is just an active database. In contrast, the network service module turns configuration entries into actual NDK objects. The service module can be altered to fit a particular need. This likely involves the creation of custom configuration tags for the configuration system. However, a full understanding of the code in NETSRV requires a basic understanding of nearly all the API functions discussed in the *TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524).

You should be most concerned about the *NetScheduler()* function because this scheduler runs the NDK. It looks for events that need to be processed by the NDK, and it performs the work necessary to start processing.

4.2 NETCTRL Scheduler

4.2.1 Scheduler Overview

The NETCTRL scheduler code is an infinite loop function named *NetScheduler()* and appears at the end of the source file NETCTRL.C. It looks for activity events from the low level device drivers, and acts when events are detected. The loop terminates when a static variable is set through an outside call to *NC_NetStop()*.

Although the NDK provides a reentrant environment, the core of the stack is not reentrant. Portions of the code must be protected from access by reentrant calls. Instead of using critical sections that block out all other Task execution, the software defines an operating mode called kernel mode. Kernel mode is defined such that only one Task may be in kernel mode at any given time. It does nothing to prevent Tasks from running that do not use the NDK. This provides protection for the stack, without affecting the execution of unrelated code. There are two functions defined to enter and exit kernel mode, *llEnter()* and *llExit()*. They are part of the OS adaptation layer, and are discussed in more detail in Section 5.2.3. In short, *llEnter()* must be called before calling into the stack, and *llExit()* must be called when done calling stack functions.

The basic flow of the scheduler loop can be summarized by this pseudo code:

```
static void NetScheduler()
{
    SetSchedulingPriority();

    while( !NetHaltFlag )
    {
        WaitOrPollForEvents();
        ServiceDeviceDrivers();

        // Process current events in Kernel Mode
        if( StackEvents )
        {
            // Enter Kernel Mode
            llEnter();
            ServiceStackEvents();

            // Exit Kernel Mode llExit();
        }
    }
}
```

The sections that follow address each of the highlighted functions in turn. Note that the code continues to run until the NetHaltFlag is set. This flag is set when an application calls the *NC_NetStop()* function.

4.2.2 Scheduling Options

There are three basic ways to run the scheduler. They can be viewed as three operating modes:

1. Scheduler runs at low priority and only when there are network events to process.
2. Scheduler runs continuously at low priority, polling the device drivers for events.
3. Scheduler runs a high priority, but only when there are network events to process.

The best way to run the scheduler depends on the application and system architecture.

Mode 1 is the most efficient way to run the NDK. Here, the scheduler loop runs at a low priority. This allows applications that potentially have real-time requirements to have priority over networking where the real-time restrictions are more relaxed. In addition, the scheduling loop only runs when there is network related activity; therefore, a standard SYS/BIOS idle loop can also be used.

Mode 2 is used when the device drivers are prevented from using interrupts. This is best for real-time Tasks, but worst for network performance. Since the scheduler thread runs continuously, it also prevents the use of a SYS/BIOS idle loop. This is the mode that NETCTRL must use when using a device driver that requires polling.

Mode 3 is the most Unix-like environment. Here, the network scheduler Task runs at a higher priority than any other networking Task in the system. The stack runs whenever new network related events are detected, pre-empting other Tasks from potentially using the stack. This is the best method for keeping the networking environment up to date without placing restrictions on how network applications are written.

Setting priority and polling or interrupt driven scheduling is done when the application first calls *NC_SystemOpen()*. This is discussed further in Section 3.2.5.2 and in the *NDK Programmer's Reference Guide*.

4.2.3 Scheduler Thread Priority

The first lines of the actual implementation of *NetScheduler()* include the following code:

```
// Set the scheduler priority
TaskSetPri(TaskSelf(), SchedulerPriority);
```

This code changes the priority of the Task thread that calls into *NC_NetStart()*, so that there is a single control point to set the scheduler priority. The priority used is that which was passed to the *NC_SystemOpen()* function. This is discussed further in Section 3.2.5.2 and in the *NDK Programmer's Reference Guide*.

The scheduler priority (relative to network application thread priority) affects how network applications can be programmed. For example, when running the scheduler in low priority, a network application cannot poll for data by continuously calling recv() in a non-blocking fashion. This is because if the application thread never blocks, the network scheduler thread never runs, and incoming packets are never processed by the NDK.

4.2.4 Tracking Events with STKEVENT

As previously mentioned, the NETCTRL module is the interface between the stack and the device drivers in the HAL layer. In older versions of the NDK, device drivers signaled the NETCTRL module through a global Semaphore. In order to improve this process slightly, the simple Semaphore has been encapsulated into an object called a STKEVENT.

From the device driver's point of view, this event object is a handle that is passed to a function called *STKEVENT_signal()*. In reality, this function is only a MACRO that operates on a structure of type STKEVENT. The NETCTRL module operates directly on this structure. The STKEVENT structure is defined as follows:

```
// Stack Event Object
typedef struct _stkevent {
    HANDLE hSemEvent;
    uint   EventCodes[3];
} STKEVENT;

#define STKEVENT_TIMER 0
#define STKEVENT_ETHERNET 1
#define STKEVENT_SERIAL 2
```

There are two parts to the structure, a Semaphore handle and an array of events. Each driver signals an event by setting a flag in the EventCode[ ] array for its event type, and then optionally signaling the event semaphore. The semaphore is only signaled when the driver detects an interrupt condition. If the event is detected during driver polling (either periodic polling or constant in the case of a polling only driver), the event is set, but the semaphore is not signaled.

Note that in a polling environment, the semaphore handle *hSemEvent* is NULL.

The NETCTRL module creates a private instance of the STKEVENT structure that it passes to device drivers as a handle of type STKEVENT_Handle. The private instance that is operated on directly by NETCTRL is declared as:

```
// Static Event Object
static STKEVENT stkEvent;
```

In the full source to NetScheduler() that follows, the STKEVENT structure is referred to by its instance *stkEvent*.

4.2.5 Scheduler Loop Source Code

The code for the example scheduler implementation included in the NDK is shown below. This implementation fleshes out the pseudo code shown in Section 4.2.1, using the methods and objects described in this section. In this code, the number of serial port devices and Ethernet devices is passed in as calling arguments. This device count is obtained from the device drivers when they are asked to enumerate their physical devices.

```
#define FLAG_EVENT_TIMER 1
#define FLAG_EVENT_ETHERNET 2
#define FLAG_EVENT_SERIAL 4

static void NetScheduler( uint const SerialCnt, uint const EtherCnt )
{
    register int fEvents;

    // Set the scheduler priority
    TaskSetPri(TaskSelf(), SchedulerPriority);

    // Enter scheduling loop
    while( !NetHaltFlag )
    {
        if( stkEvent.hSemEvent )
        {
            SemPend( stkEvent.hSemEvent, SEM_FOREVER );
            SemReset( stkEvent.hSemEvent, 0 );
        }

        // Clear our event flags
        fEvents = 0;

        // First we do driver polling. This is done from outside
        // kernel mode since pure "polling" drivers can not spend
        // 100% of their time in kernel mode.

        // Check for a timer event and flag it
        if( stkEvent.EventCodes[STKEVENT_TIMER] )
        {
            stkEvent.EventCodes[STKEVENT_TIMER] = 0;
            fEvents |= FLAG_EVENT_TIMER;
        }

        // Poll only once every timer event for ISR based drivers,
        // and continuously for polling drivers. Note that "fEvents"
        // can only be set to FLAG_EVENT_TIMER at this point.
        if( fEvents || !stkEvent.hSemEvent )
        {
#ifndef _INCLUDE_NIMU_CODE
            // Poll Ethernet Packet Devices
            if( EtherCnt )
                _llPacketServiceCheck( fEvents );
#else
            NIMUPacketServiceCheck (fEvents);
#endif

            // Poll Serial Port Devices
            if( SerialCnt )
                _llSerialServiceCheck( fEvents );
        }

        //
        // Note we check for Ethernet and Serial events after
        // polling since the ServiceCheck() functions may
        // have passively set them.
        //
```

```
#ifndef _INCLUDE_NIMU_CODE
        // Check for a Ethernet event and flag it
        if(EtherCnt && stkEvent.EventCodes[STKEVENT_ETHERNET] )
        {
            // We call service check on an event to allow the
            // driver to do any processing outside of kernel
            // mode that it requires, but don't call it if we
            // already called it due to a timer event or by polling
            if( !(fEvents & FLAG_EVENT_TIMER) && stkEvent.hSemEvent )
                _llPacketServiceCheck( 0 );

            // Clear the event and record it in our flags
            stkEvent.EventCodes[STKEVENT_ETHERNET] = 0;
            fEvents |= FLAG_EVENT_ETHERNET;
        }

#else
        /* Was an Ethernet event signaled? */
        if(stkEvent.EventCodes[STKEVENT_ETHERNET])
        {
            // We call service check on an event to allow the
            // driver to do any processing outside of kernel
            // mode that it requires, but don't call it if we
            // already called it due to a timer event or by polling
            if (!(fEvents & FLAG_EVENT_TIMER) && stkEvent.hSemEvent)
                NIMUPacketServiceCheck (0);

            // Clear the event and record it in our flags
            stkEvent.EventCodes[STKEVENT_ETHERNET] = 0;
            fEvents |= FLAG_EVENT_ETHERNET;
        }

#endif  /* _INCLUDE_NIMU_CODE */
        // Check for a Serial event and flag it
        if(SerialCnt && stkEvent.EventCodes[STKEVENT_SERIAL] )
        {
            // We call service check on an event to allow the
            // driver to do any processing outside of kernel
            // mode that it requires, but don't call it if we
            // already called it due to a timer event or by polling
            if( !(fEvents & FLAG_EVENT_TIMER) && stkEvent.hSemEvent )
                _llSerialServiceCheck( 0 );

            // Clear the event and record it in our flags
            stkEvent.EventCodes[STKEVENT_SERIAL] = 0;
            fEvents |= FLAG_EVENT_SERIAL;
        }

        // Process current events in Kernel Mode
        if( fEvents )
        {
            // Enter Kernel Mode
            llEnter();

            // Check for timer event
            if( fEvents & FLAG_EVENT_TIMER )
                ExecTimer();

#ifndef _INCLUDE_NIMU_CODE
            // Check for packet event
            if( fEvents & FLAG_EVENT_ETHERNET )
                llPacketService();
#else
            // Check for packet event
            if( fEvents & FLAG_EVENT_ETHERNET )
                NIMUPacketService();
```

```
#endif
            // Check for serial port event
            if( fEvents & FLAG_EVENT_SERIAL )
                llSerialService();

            // Exit Kernel Mode
            llExit();
        }
    }
}
```

4.3 Disabling On-Demand Services

The NETCTRL library is designed to support "potential" stack features that the user may desire within an application (e.g. DHCP server). However, the drawback of this is that the code for such features will be included in the executable even if the application never uses the features. This results in a larger footprint than is usually necessary. To minimize this problem, the following different versions of the NETCTRL library are available:

- **netctrl_min.** This minimal library enables only the DHCP client. It should be used when a minimal footprint is desired.
- **netctrl.** This "standard" version of the NETCTRL library enables the following features and has a medium footprint:
  - Telnet server
  - HTTP server
  - DHCP client
- **netctrl_full.** This "full" library enables all supported NETCTRL features, which include:
  - Telnet server
  - HTTP server
  - NAT server
  - DHCP client
  - DHCP server
  - DNS server

Each of these NETCTRL library versions is built for both pure IPv4 as well as IPv6.

If you configure the NDK in CCS with the XGCONF configuration tool, the appropriate NETCTRL library is automatically selected based on the modules you enable.

If you need even more control over which features are available in the NETCTRL library used by your application, you can #define the following constants in <NDK_INSTALL_DIR>\packages\ti\ndk\inc\netctrl\netsrv.h, which control the features brought into the NETCTRL library if "_NDK_EXTERN_CONFIG" is not defined.

```
#define NETSRV_ENABLE_TELNET 1
#define NETSRV_ENABLE_HTTP 1
#define NETSRV_ENABLE_NAT 0
#define NETSRV_ENABLE_DHCPCLIENT 1
#define NETSRV_ENABLE_DHCPSERVER 1
#define NETSRV_ENABLE_DNSSERVER 1
```

By setting any of the above to 0 and rebuilding the appropriate NETCTRL library, individual services can be purged from the executable.

OS Adaptation Layer : OS and MiniPrintf

The OS adaptation layer controls how the NDK uses SYS/BIOS resources. This includes Tasks, Semaphores, memory and printing. Anything OS related can be adjusted here. This chapter also includes a history of the OS adaptation layer source, and describes the files which comprise the source code.

Topic **Page**

5.1 Introduction to OS Source

5.1.1 History

One reason the NDK contains an OS adaptation layer is so that applications that are coded to the abstraction can be executed in any environment to which the abstraction is ported. For hardware-centric applications, cross-platform portability is not usually practical nor required. Embedded programmers often prefer to use SYS/BIOS and take advantage of the support features provided by Code Composer Studio.

For most of the OS adaptation API, the abstraction functions are converted to direct SYS/BIOS calls through the use of #define macros. However, there are some additions and refinements made at the OS layer that tend to vary slightly from one SYS/BIOS based system to the next. For these refinements, external OS abstraction functions are required. This allows the system programmer to adapt the OS layer to meet the particular system requirements of their SYS/BIOS based environment.

This section covers the OS functions that may need to be adjusted. The OS source code referenced in this section is found in the SRC\OS directory. Printing functions have been separated into a different directory and library; this way, if you need to use the bigger size, more feature rich, printing APIs provided with the standard RTS library, you can easily replace the slim APIs in MiniPrintf with the RTS API.

5.1.2 Source Files

Source code to the OS library consists of several files, located in the <NDK_INSTALL_DIR>\packages\ti\ndk\os directory:

efs.c	Embedded (RAM based) file system
intmgmt.c	OS abstraction wrapper for configuring interrupts using underlying OS
mem.c	Memory allocation and memory copy functions
mem_data.c	Memory allocation definitions and #pragmas
oem.c	OEM Cache and System functions
ossys.c	Additional OS support (debug logging, stricmp() function)
semaphore.c	Semaphore abstraction
task.c	Task thread abstraction

The printing functionality is contained in the following file, located in the <NDK_INSTALL_DIR>\packages\ti\ndk\miniPrintf directory:

miniprintf.c	Basic printf() functions

Two additional include files are located in the <NDK_INSTALL_DIR>\packages\ti\ndk\inc\os directory:

osif.h	Interface specifications to the adaptation library
oskern.h	Semi-private declarations for use by functions like NETCTRL

5.2 Task Thread Abstraction: TASK.C

The TASK.C module contains a subset of the Task abstraction API documented in the *TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524). It also contains the source code to the stack's exclusion method functions: *llEnter()* and *llExit()*. The latter are discussed in Section 5.2.3 of this document.

Most of the Task and Semaphore functions defined in the *TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524) are in actuality macros that call SYS/BIOS. These macros are defined in INC\OS\OSIF.H. The functions that do not directly map to SYS/BIOS are listed here.

5.2.1 TaskSetEnv() and TaskGetEnv()

The set environment and get environment functions are supplied in task.c so that they can be ported to the SYS/BIOS based system in such a way that they do not conflict with other system use of the *Task_setEnv()* and *Task_getEnv()* functions.

The ability to associate a data structure with a Task thread is essential for the stack library. The problem with the implementation in SYS/BIOS is that it only allows a single entity to assign this environment pointer. The result is that any use of *Task_setEnv()* or *Task_getEnv()* by a third party conflicts with the stack software.

The implementation of the task.c supplied in the NDK gets around this limitation by using the SYS/BIOS Task object hook functions. Sets of hook functions allow multiple functions to hook into SYS/BIOS Task creation—including the ability to expand the environment. For more information, see the SYS/BIOS documentation.

NOTE: In the task.c module provided, the *TaskSetEnv()* and *TaskGetEnv()* functions return without setting or getting an environment variable if the "slot" parameter is non-zero. All internal stack functions use slot zero. The additional slots were originally indented to be used by other operating systems. Under SYS/BIOS, applications must use *Task_setEnv()* and *Task_getEnv()* functions. Therefore, the SYS/BIOS based implementation of *TaskSetEnv()* and *TaskGetEnv()* is simplified.

5.2.2 TaskCreate(), TaskExit(), and TaskDestroy()

The create, exit and destroy functions all call their SYS/BIOS equivalents.

For *TaskExit()* and *TaskDestroy()* to function as expected, the SYS/BIOS Task.deleteTerminatedTasks configuration parameter must be set to “true”. This parameter setting instructs the Task module to delete completed Tasks in the SYS/BIOS Idle task. Part of cleaning up involves freeing Task stack memory.

The NDK configuration sets the Task.deleteTerminatedTasks to “true” automatically. If your application does not use the NDK Global module in XGCONF for configuration, it must manually set this parameter. For example:

```
var Task = xdc.useModule('ti.sysbios.knl.Task');
Task.deleteTerminatedTasks = true;
```

NOTE: In previous releases, the NDK did not require this configuration setting. If this configuration setting is not made, Task clean-up will not occur, and out of memory errors may occur because the task stacks and objects will not be freed when *TaskExit()* and/or *TaskDestroy()* is called.

5.2.3 Choosing the llEnter()/llExit() Exclusion Method

Although the NDK provides a reentrant environment, the core of the stack is not reentrant. Portions of the code must be protected from access by reentrant calls. Instead of using critical sections that block out all other Task execution, the software defines an operating mode called kernel mode. Kernel mode is defined such that only one Task may be in kernel mode at any given time. It does nothing to prevent Tasks from running that do not use the NDK. This provides protection for the stack software, without affecting the execution of unrelated code.

The *llEnter()* and *llExit()* functions are used throughout the stack code to enter and exit kernel mode, and provide code exclusion without using critical sectioning. They are equivalent to the *splhigh()*/*splx()* Unix functions and their multiple cousins.

There are two example implementations of the *llEnter()* and *llExit()* functions included in the NDK. The example implementations provide exclusion through Task priority or by using Semaphores. Source code to both implementations is included in the Task abstraction source file: `SRC\OS\TASK.C`

One method of exclusion is the priority method. Here, the Task that calls *llEnter()* is boosted to a priority level of OS_TASKPRIKERN, which guarantees that it will not be pre-empted since it is impossible for another Task to be running (all Tasks that can possibly call into the stack have a lower priority level). The stack is coded so that a Task at the kernel mode priority level will never block. When *llExit()* is called, the Task's original priority is restored. Note that time critical Tasks can be assigned a priority higher than OS_TASKPRIKERN, but they are not allowed to call into the NDK.

Priority-based exclusion makes it important that your application use only the *TaskCreate()* API combined with one of the NDK defined task priorities. If you call the SYS/BIOS *Task_create()* API with a priority greater than NDK's kernel priority level, the priority-based exclusion is likely to break. Setting a thread to a higher priority than the NDK's high-priority thread level may disrupt the system and cause unpredictable behavior if the thread calls any stack-related functions.

An alternate implementation of the enter and exit functions uses a Semaphore with an initial count of 1. When *llEnter()* is called, the Task calls a pend operation on the Semaphore. If some other Task is currently executing in kernel mode, the new Task will pend until *llExit()* is called by the original Task. A call to *llExit()* results in a post operation which frees up one Task to enter the stack. This form of the function pair is safer than the priority method, but may also be slower. In general, Semaphore operations are a little slower than Task priority changes. However, this method also has its advantages. The main advantage with the Semaphore method is that Tasks can be assigned priority levels more freely. There is no need to restrict Task priority or be concerned if a high priority Task is going to call into the NDK.

By altering the #if statements around the two implementations, the system developer can choose to use either implementation.

5.3 Packer Buffer Manager: PBM.C

The Packet Buffer Manager (PBM) is charged with managing all the packet buffers in the system. Packet buffers are used by the NDK and device drivers to carry networking packet data. The PBM programming abstraction is discussed in the *NDK Programmer's Reference Guide*. This section discusses the implementation provided in the NDK.

5.3.1 Packet Buffer Pool

The PBM buffers are configured in the **Buffers** tab of the Global module configuration in XGCONF. You can set the **Number of frames** (default = 192), the **Frame buffer size** (default=1536 bytes), and the memory section where the buffers are stored.

Note that when the memory is declared, it is placed on a cache aligned boundary. Also, each packet buffer must be an even number of cache lines in size so that it can be reliably flushed without the risk of conflicting with other buffers.

If you *are not* using Macronix (LogicIO) Ethernet, you can set the Frame buffer size to 1536.

If you *are* using Macronix (LogicIO) Ethernet, a 1664-byte packet buffer is needed. In a very simple Ethernet system, the max size of the frame buffer would be 1518, but a few things in the NDK environment change that size. First, all devices use a standard packet header size from the start of the packet buffer to the IP header. The result is that any packet can be routed to any device without altering the location of the IP header. The standard size used is 22 bytes, which is the size of a PPPoE header plus the standard Ethernet header. Next, the Macronix Ethernet MAC transfers data in 16-byte bursts. However, the first four bytes of the first transfer is 4 bytes of status, leaving only 12 bytes of data. A little math reveals the Macronix writes 1532 bytes into the packet buffer on a 1518 byte frame. Taking the standard 1532 bytes required by Macronix, and adding an additional 8 byte pad for the standard header (22 – standard 14 byte Ethernet) gives 1540 bytes which rounds to 1664 when expanded to fill a full L2 cache line.

5.3.2 Packet Buffer Allocation Method

The basic method of buffer allocation is the buffer pool. Buffers are allocated when the *PBM_alloc()* function is called. This function can be called at interrupt time, so you must ensure only non-blocking calls are made as a result. However, only device drivers can make calls from an ISR and device drivers never ask for a buffer larger than PKT_SIZE_FRAMEBUF. Therefore, the fallback method for allocating larger buffers can technically make blocking calls, although the implementation included in the NDK does not make blocking calls under any circumstance.

The basic method of allocation is to check the size. When the size is less than or equal to PKT_SIZE_FRAMEBUF, then the packet buffer is obtained off the free queue. If there are no free packet buffers on the queue, the function returns NULL. Note that the PBM module could be modified to grow the free pool or use memory allocation as a fallback, but any buffer supplied as a result of a request with the size less than or equal to PKT_SIZE_FRAMEBUF, must adhere to the cache line restrictions outlined in the previous section.

For packet buffers larger than PKT_SIZE_FRAMEBUF, standard memory can be used. These allocation requests are only made for re-assembling large IP packets. The resulting packet cannot be submitted to a hardware device without being fragmented. Therefore, the packet buffer does not need to be compatible for hardware transmission.

5.3.3 Referenced Route Handles

One of the fields in the PBM structure is a referenced handle to a route used to route a packet to its final destination. The PBM module must be aware of this handle when freeing a packet buffer or copying a packet buffer.

When packet buffer is freed by calling *PBM_free()*, the PBM module must check for a route handle held by the packet buffer, and dereference the handle if it exists. For example:

```
if( pPkt->hRoute )
{
    RtDeRef( pPkt->hRoute );
    pPkt->hRoute = 0;
}
```

As noted in the source code to PBM.C, the function *RtDeRef()* can only be called from kernel mode. However, instead of defining two versions of the *PBM_free()* function, the PBM module relies on the fact that device drivers are never given packet buffers containing routes. Therefore, any call to *PBM_free()* where the buffer contains a route, must have been called from within kernel mode. It is, therefore, safe to call *RtDeRef()*.

When a packet buffer is copied with *PBM_copy()*, all the information about the packet is also copied. This information may include a referenced route handle. If the handle to a route is copied in the process of copying the packet buffer, then a reference to that handle must also be added by calling the *RtRef()* function. The PBM module does not need to worry about kernel mode for the same reason as it did not with *PBM_free()*.

5.4 Memory Allocation System: MEM.C

The memory allocation system consists of allocation functions for small blocks of memory, large blocks, and for initializing and copying memory blocks. The API definitions for the files contained in this module is defined in the *TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524). These functions are used throughout the stack. The source code is provided so the systems programmer can adapt the memory system to fit a particular need.

The size and placement of this Memory Manager Buffer array can be configured in the **Buffers** tab of the Global module configuration in XGCONF. The **Page size** is depended upon by various stack entities, so you should be careful when changing it. The **Number of pages** used can be adjusted up or down to increase or decrease the scratchpad memory size.

The allocation functions for the small memory blocks (*mmAlloc()* and *mmFree()*) should not be altered. These functions are used by the NDK to allocate and free scratchpad type memory. They can be called at interrupt time and are not allowed to block. The memory is currently allocated out of a static array.

The memory manipulation functions *mmZeroInit()* and *mmCopy()* are both coded in C. A system programmer may recode these functions in assembly, or to use an EDMA channel to move memory.

The allocation functions for the large memory blocks (*mmBulkAlloc()* and *mmBulkFree()*) are currently defined to use *Memory_alloc()* and *Memory_free()* on the default heap. These functions can be altered to use any memory allocation system of choice. They are not called at interrupt time and are allowed to block.

The default Heap is used for allocations. If you want to change the Heap used by *mmBulkAlloc()*, you must use SYS/BIOS or XDCtools APIs as appropriate.

5.5 Embedded File System: EFS.C

The EFS file system provides RAM based file support for the HTTP server and any CGI functions provided by the applications programmer. This API is defined in the *TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524). The source code is provided for adapting the functions to support a physical storage media. This allows the HTTP server to work on the physical device without porting the server.

5.6 General OS Support: OSSYS.C

The OSSYS file is a generic catch-all for functions that do not have a home elsewhere. Currently, this module contains *DbgPrintf()*—a debug logging function and *stricmp()*, which is not contained in the RTS.

5.7 Print Functions: MINIPRINTF.C

The MINIPRINTF.C module in the <NDK_INSTALL_DIR>\packages\ti\ndk\MiniPrintf directory contains an implementation of *printf()*, *sprintf()*, *vprintf()*, and *vsprintf()*. These are basic implementations that do not support floating point. The function at the top of the module, *printstr()* can be altered to redirect standard output from the debugger to a buffer or some external device.

5.8 Jumbo Packet Buffer Manager (Jumbo PBM)

The jumbo packet buffer manager is responsible for handling memory allocation and de-allocation for packet buffers of size greater than MMALLOC_MAXSIZE (3068 bytes). This packet buffer manager is useful when the application intends on using Jumbo frames, i.e., packets larger than 1500 bytes in size that cannot be typically allocated using PBM or SYS/BIOS APIs in an interrupt context.

The following are some of the main features of the Jumbo PBM:

- The Jumbo PBM implementation is by large similar to the PBM implementation itself, except for the block sizes it can handle are larger than the ones in PBM and ranges between 3K and 10K bytes by default.
- Jumbo PBM does not use any SYS/BIOS APIs or dynamic memory allocation method for it's memory allocation and thus can be used safely in interrupt context. It uses a static memory allocation method, i.e. it reserves a chunk of memory in the "far" section of the device memory and it further uses it to allocate for the packet buffers required.
- The Jumbo PBM allocates memory off a separate section in the memory than the PBM itself. PBM uses the memory sections "NDK_PACKETMEM", "NDK_MMBUFFER" for its memory allocation. On the other hand, Jumbo PBM defines and uses a section called "NDK_JMMBUFFER" for its memory allocation. The size of this section and its placement are all customizable.
- A sample implementation of the Jumbo PBM is provided in the NDK OS AL. The customer is expected to customize this implementation according to their application needs and system's memory constraints. The memory section sizes, block sizes and the allocation method itself is all up for customization.
- Jumbo PBM APIs are not expected to be invoked directly. The application and driver must call the PBM_alloc() / PBM_free() APIs only. These APIs in turn invoke the Jumbo PBM APIs to allocate/clean-up memory if the memory requested is larger than what PBM itself can handle, i.e., 3K bytes.

For a sample implementation of the Jumbo PBM please refer to the source file JUMBO_PBM.C in the <NDK_INSTALL_DIR>\packages\ti\ndk\stack\pbm directory.

5.9 Interrupt Manager

The Interrupt Manager defines the APIs and data structures required to configure and manage interrupts in a generic way. This wrapper hides the OS specific implementation details of interrupt management by providing a unified API to do the same.

The NDK interrupt manager implementation provided in this package uses SYS/BIOS as its underlying OS. It uses the SYS/BIOS Hwi module (and the EventCombiner module for C6000 targets) to configure the interrupts.

The APIs and data structures exported by this module are documented in the *TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524). Sample implementations using these APIs can be found in any of the Ethernet driver's code of the supported platform's NDK Support Packages.

Revision History

Table A-1 lists the changes made since the previous version of this document.

Table A-1. Document Revision History

Reference	Additions/Modifications/Deletions
	Change Summary for NDK 2.21
Section 1.7 and Section 3.1	New graphical configuration support for NDK modules has been added within the XGCONF configuration tool in CCS. Many settings that had to be made in C code can now be made in the configuration.
Section 1.1	In addition to C6000 support, the NDK now supports the Cortex-A8 and ARM9 in ELF format.
Section 1.6	The NDK is still compatible with Code Composer Studio (CCS) 4.2, but it is suggested that you use it with CCS 5.1+.
Section 1.3.3 and Section 1.5.1	The directory structure for the NDK has changed. Libraries and source files are stored together in a separate directory named for each library, rather than in a \lib and \src directory.
Section 1.3.8	There are three choices for the network control library; this allows you to eliminate unused stack features from the build.
Section 1.2 and Section 1.5.5	Examples are no longer shipped with the NDK core software product. Instead, they are provided in separate NDK Support Package (NSP) products or as part of the SDK for that device family.
Section 1.5.4	Test applications are now provided for Linux (in addition to Windows).
	DSP/BIOS 5.x is no longer supported; use the NDK with SYS/BIOS 6.30+. Changes to example code have been made throughout this document to use SYS/BIOS modules in place of DSP/BIOS modules.
	Changes in Earlier Versions
Related Documentation From Texas Instruments	Added new document reference
Section 1.3.1	Added/replaced text
Section 1.3.2	Added text
Section 1.3.4	Added/replaced text
Section 1.3.5	Added text
Section 1.3.8	Added/replaced text
Section 1.6.3	New section/replaced text
	Deleted CCStudio Project Link Order section.
Section 3.5.1	Changed "MMALLOC_SIZE" to MMALLOC_MAXSIZE"
	Changed CFGITEMP_IP_IPREASMMAXSIZE to CFGITEM_IP_IPREASMMAXSIZE
	Changed CFGITEM_IP_IPSOCKUDPRXLIMIT to CFGITEM_IP_SOCKUDPRXLIMIT
Section 4.2.5	Replaced code
Section 1.2.1	Replaced text
Section 1.3.6	Replaced text
Section 1.4.5	Replaced text
Section 1.5	Replaced/added text
Section 1.5.3	Replaced text
	Deleted previous Building in NIMU section.
Section 2.2.3.4	Added new section
Section 3.4	Added note
Section 5.1.2	Added text

Table A-1. Document Revision History (continued)

Reference	Additions/Modifications/Deletions
Section 5.8	Added new section
Section 5.9	Added new section