NDK User's Guide

Table of Contents

1 Overview

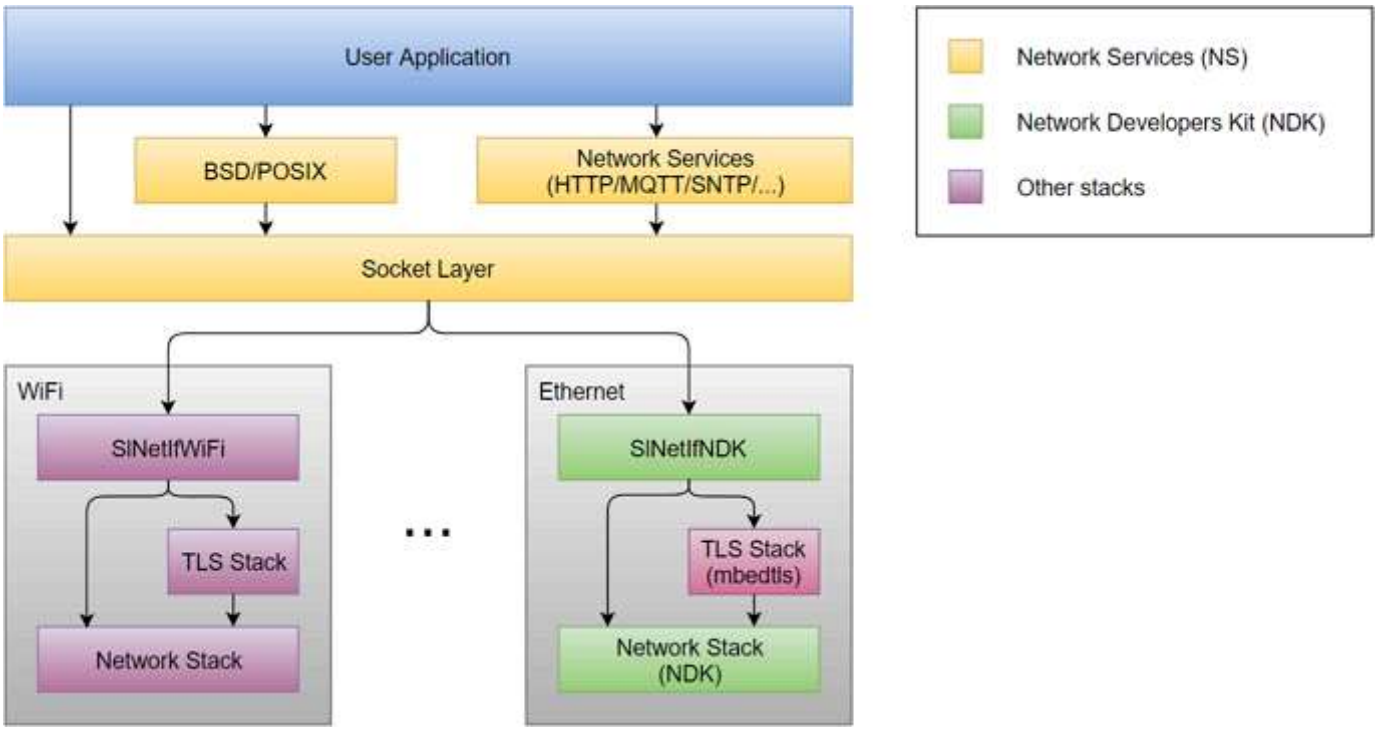
This chapter introduces the Network Developer's Kit (NDK) by providing a brief overview of the purpose and construction of the NDK, along with hardware and software environment specifics in the context of NDK deployment. This Network Developer's Kit (NDK) Software User's Guide serves as an introduction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

1.1 Introduction

The Network Developer's Kit (NDK) is a platform for development and demonstration of network enabled applications on TI embedded processors. The code included in this NDK release is generic C code which runs on a variety of TI devices.

The NDK is typically paired with the Network Services (NS) Component. NS provides a stack independent implementation of BSD sockets, as well as higher level services (e.g. HTTP, SNTP). For more information, see the NS documentation.

Although not strictly precise (e.g. applications will call NDK APIs directly as well), the following diagram shows how NS and the NDK typically relate to each other in a complete system.



In the figure above, the user application can make calls using the standard BSD sockets APIs, or it can directly call into the SlNetSock layer to communicate with a network connection. The SlNetSock layer is a stack-independent layer between the user application and the service-specific stacks.

Within an SDK, the Network Services SlNetSock module utilizes the NDK as the network stack for wired Ethernet communications. The "SlNetIfNDK" is the implementation of the SlNetSock interface for the NDK (and can be found in the NDK's `/ti/ndk/slnetif` directory).

The NDK stack's settings may be configured at run time by making calls to the NDK's Cfg*() functions.

The NDK is designed to provide a platform-independent, device-independent, and RTOS-independent API interface for use by the application. Many of the API modules described in this document and in the NDK API Reference Guide are rarely used by applications. Instead they are available to those who are writing device drivers and network stacks.

A user application typically uses the following APIs provided by the NDK:

- **Cfg functions** add settings to the configuration database that determine which network services will be available to the application. For details, see the NDK API Reference Guide.
- **NC_ functions** cause the network services system to be initialized, started, and stopped. For details, see the NDK API Reference Guide.
- **TaskCreate() or native OS functions** to handle application threading. The NDK's TaskCreate() API can be used to create a thread for any RTOS supported for the target device by the SimpleLink SDK. For most targets, this includes TI-RTOS Kernel and FreeRTOS. Alternatively, the native thread creation APIs can be used for the supported RTOS being used. For details, see "Creating a Task".
- **NDK socket APIs** to perform socket actions such as accept, send, and receive. For a pure NDK application, these are the BSD-like NDK_*() functions. But typical applications should use the standard BSD APIs provided by NS. For details on the BSD-like NDK APIs, see the NDK API Reference Guide.

1.2 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 Using gmake topic in the Texas Instruments Wiki.

You can define the following macros to cause variations in the behavior of the rebuilt NDK libraries:

- _INCLUDE_IPv6_CODE – If defined, IPv6 support is enabled.
- _INCLUDE_JUMBOFRAME_SUPPORT – If defined, packet sizes larger than 1500 bytes are supported.
- _INCLUDE_NAT_CODE – If defined, Network Address Translation (NAT) support is provided by the stack.
- _INCLUDE_PPP_CODE – If defined, the Point-to-Point Protocol (PPP) module is included.
- _INCLUDE_PPPOE_CODE – If defined, enable the PPP over Ethernet (PPPoE) client.
- _STRONG_CHECKING – If defined, error checking is performed on all handles.

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 the RTOS and to the system peripherals.

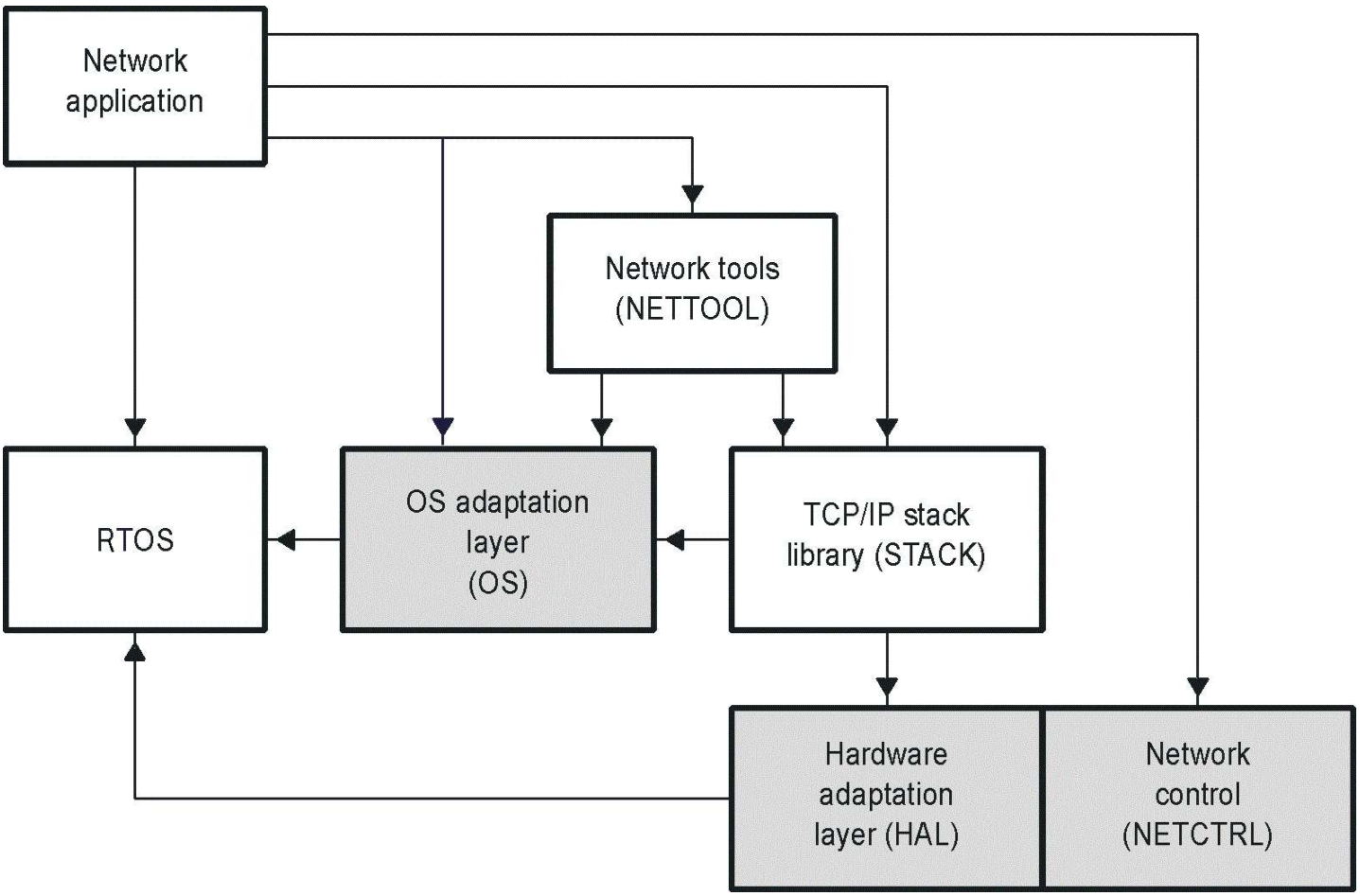
The features of the NDK include the following:

- **Virtual LAN (VLAN) support.** VLAN support enables the stack to receive, process, and transmit VLAN tagged packets.
- **Raw Ethernet socket support.** Raw Ethernet sockets (different from Raw IPv4/IPv6 sockets) enable 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).
- **IPv6 stack support.** The stack is available for both IPv6 and IPv4.
- **Jumbo frames support.** Jumbo frames have packet sizes larger than 1500 bytes. 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 NDK API Reference Guide.

1.3.2 Control Flow

The following figure 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, 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).



1.3.3 Library Directory Structure

> **NOTE:** The location of NDK files referenced in this document will vary depending on the type of installation you performed to obtain the NDK component. If you installed a SimpleLink SDK, the files are likely to be located in your `<SDK_INSTALL_DIR>/sources/ti/ndk` directory. If you installed some other type of SDK, the files are likely to be located in your `<NDK_INSTALL_DIR>/packages/ti/ndk` directory. For simplicity, this document refers to the directories beginning at the `/ti/ndk` level, which is common to all installations.

Pre-built linkable libraries and source code are provided for each of the libraries that make up the NDK in the `/ti/ndk` directory tree. The pre-built libraries are in a lib subdirectory of the directory for each library. See "NDK Software Directory Structure" for more about the directory tree.

- Both IPv4 and IPv6 libraries are provided. Filenames that do not include "ipv4" are compiled for IPv6. However, in the `/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 "NETCTRL Libraries" for details.

- The OS library comes in TI-RTOS Kernel ("os" and "os_sem") and FreeRTOS ("os_freertos" and "os_sem_freertos") versions.
- 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 include the following:

Extension	Description
.aa8fg	For Cortex-A8 targets (ELF format, little endian, GNU compiler)
.aa9fg	For Cortex-A9 targets (ELF format, little endian, GNU compiler)
.aa15fg	For Cortex-A15 targets (ELF format, little endian, GNU compiler)
.ae9	For ARM9 targets (ELF format, little endian, TI compiler)
.ae66	For C66x targets (ELF format, little endian, TI compiler)
.ae674	For C674x targets (ELF format, little endian, TI compiler)
.aea8f	For Cortex-A8 targets (ELF format, little endian, TI compiler, does not use hardware-based floating point support, for legacy application support)
.aea8fnv	For Cortex-A8 targets (ELF format, little endian, TI compiler, uses hardware-based vector floating point support, recommended over .aea8f)
.aem3	For Cortex-M3 targets (ELF format, little endian, TI compiler)
.am3g	For Cortex-M3 targets (ELF format, little endian, GNU compiler)
.arm3	For Cortex-M3 targets (ELF format, little endian, IAR compiler)
.aem4	For Cortex-M4 targets (ELF format, little endian, TI compiler)
.am4g	For Cortex-M4 targets (ELF format, little endian, GNU compiler)
.arm4	For Cortex-M4 targets (ELF format, little endian, IAR compiler)
.aem4f	For Cortex-M4F floating point targets (ELF format, little endian, TI compiler)
.am4fg	For Cortex-M4F floating point targets (ELF format, little endian, GNU compiler)
.arm4f	For Cortex-M4F floating point targets (ELF format, little endian, IAR compiler)

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 RTOS, 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 `/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.

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 `/ti/ndk/nettools/lib` directory.

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

See "NETTOOL Services and Support Functions" for more information.

1.3.6 OS Library

These libraries form a thin adaptation layer that maps some abstracted OS function calls to POSIX and native OS API calls. This includes Task thread management, memory allocation, packet buffer management, printing, logging, critical sectioning, and jumbo packet buffer management.

The OS libraries are provided in the `/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 "Operating System Abstraction" for more information.

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 `/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	Timer Driver Using POSIX Timer object
userled_stub/lib/hal_userled_stub	User LED Stub Driver

See "Hardware Adaptation Layer API" for information about the HAL APIs. The HAL is also discussed in the NDK API Reference Guide 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:

- Initialize the NDK and low-level device drivers
- Boot and maintain system configuration via configuration service provider callback functions
- Interface to the low-level device drivers and scheduling driver events to call into the NDK
- Unload the system configuration and driver cleanup on exit
- Initialize the NIMU core during stack bring-up, which in turn initializes and starts all the device drivers registered with the NIMU core. Initialize the VLAN module in the NDK core stack.

- De-initialize the NIMU core during stack shutdown, which in turn cycles through all the registered device drivers and shuts them down.
- Poll 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 `/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 (deprecated! - migration guide found here)
  - DHCP client
- **netctrl_full.** This "full" library enables all supported NETCTRL features, which include:
  - Telnet server
  - HTTP server (deprecated! - migration guide found here)
  - 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 Code Composer Studio (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 `/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 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 NDK API Reference Guide.

1.4.1 Operating System Abstraction

The OS abstraction provides services enabling porting to other operating systems. In most cases, these are internal to the NDK stack, and application developers will not use these APIs. However there are times when application developers may need to use the Task module. See the NDK API Reference Guide for those details.

1.4.2 Sockets and Stream IO API

The sockets API primarily consists of BSD-like socket services 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 similar 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 NDK API Reference Guide and the *Network Developer's Kit (NDK) Support Package Ethernet Driver* (SPRUFP2).

1.5 NDK Software Directory Structure

The NDK files are located in `/ti/ndk` and are organized into the following subdirectories. (Any other directories are for internal use only.) For each library, both source files and pre-build libraries are provided.

Directory	Description
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 "HAL Libraries".
inc	NDK include file directory. See "NDK Include File Directory".
netctrl	Contains libraries and source code for network startup and shutdown, including special versions for various subsets of network functionality. See "NETCTRL Libraries".
nettools	Contains libraries and source code for network tools, such as DHCP, DNS, and HTTP. See "NETTOOL Services and Support Functions".
os	Contains libraries and source code for the OS Adaptation Layer. See "OS Library".
stack	Contains libraries and source code for the network stack. See "STACK Library".
tools	Contains libraries and source code for several network tools. See "Tool Programs".
winapps	Contains client test applications for Windows and Linux command-prompt use. Both source code and executables are provided. See "Windows and Linux Test Utilities".

1.5.1 NDK Include File Directory

The include file directory (`/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

Filename	Description
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.

1.5.2 Tool Programs

The NDK provides several tools for various purposes. These are located in the `/ti/ndk/tools` directory.

Subdirectory	Variants	Description
/binsrc	–	Converts HTML files to C arrays (deprecated as the HTTP Server is deprecated - please use the HTTP Server from NS - migration guide found here).
/cgi	cgi	Functions for parsing embedded HTTP Common Gateway Interface (CGI) files (deprecated as the HTTP Server is deprecated - please use the HTTP Server from NS).
/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.

1.5.3 Windows and Linux Test Utilities

> **NOTE:** These test applications are deprecated and planned for removal!

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 examples 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.

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.

1.5.4 Example Programs

Examples for use with the NDK are provided with your SDK, and will differ depending on the SDK target.

1.6 Configuring NDK Modules

To configure the NDK and its components, the NDK allows you to use either C code to call Cfg*() functions or the XGCONF configuration tool within CCStudio. Choose one method or the other to configure your application.

- **C Code.** Configure the application by writing C code that calls *CfgNew()* to create a configuration database and other Cfg*() functions to add various settings to that configuration database. Some additional configuration is done in the linker command file. For details, see Configuring the NDK with C Code. This configuration method is recommended because it can be used with any RTOS supported by the SDK.
- **XGCONF.** Use graphical displays within CCStudio to enable and set properties. XGCONF can also be used to configure objects used by the TI-RTOS Kernel. For details, see Configuring the NDK with XGCONF. This configuration method can be used only if your RTOS is the TI-RTOS Kernel.

> **NOTE:** You should not mix configuration methods. If a project uses both methods, there will be conflicts between the two configurations.

2 Network Application Development

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.

2.1 Configuring the NDK with C Code

To configure your application's use of the NDK and its components, you can use either C code or the XGCONF configuration tool within CCStudio. Choose one method or the other to configure your application.

- **C Code.** Configure the application by writing C code that calls *CfgNew()* to create a configuration database and other Cfg*() functions to add various settings to that configuration database. Some additional configuration is done in the linker command file. This configuration method is recommended because it can be used with any RTOS supported by the SDK. If you are using this configuration method, see the subsections that follow.
- **XGCONF.** Use graphical displays within CCStudio to enable and set properties. XGCONF can also be used to configure objects used by the TI-RTOS Kernel. This configuration method can be used only if your RTOS is the TI-RTOS Kernel and you are using CCStudio to develop your application. If you are using XGCONF for configuration, see Configuring the NDK with XGCONF.

> **NOTE:** Do not mix configuration methods. If a project uses both methods, there will be conflicts between the two configurations.

2.1.1 Required RTOS Objects

The NDK uses the OS adaptation layer to access the RTOS (TI-RTOS Kernel or FreeRTOS) and the HAL layer to access the hardware. These layers require the following RTOS object to be created in order for the NDK to work properly. This requirement can only be altered by modifying the code for the OS and HAL layers.

- **Timer object.** The timer driver in the HAL requires that an RTOS 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()*. See "Constructing a Configuration for a Static IP and Gateway" for an example from `ndk.c` that creates and starts a timer object.

(If you use XGCONF to configure the NDK, this object is created automatically.)

2.1.2 Include Files

If you are using the 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 NDK Include File Directory. You should include the base NDK include directory in the project build options. For example, the project should be set to use the include file path: `/ti/ndk/inc`.

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

2.1.3 Library Files

If you are using the Cfg*() functions for configuration, you are responsible for linking the correct libraries into your project. If you are using CCStudio to manage your application, you can add the desired library files directly to the CCStudio project. This way, the linker will know where to find them.

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

2.1.4 System Configuration

If you are using the Cfg*() functions for configuration, 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 begins 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 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 examples are provided in the Configuration Examples section of this document and in the NDK example programs.

2.1.4.1 Configuration Examples

This section contains some sample code for constructing configurations using the Cfg*() functions.

2.1.4.1.1 Constructing a Configuration for a Static IP and Gateway

The *ndkStackThread()* function in this example consists of the main initialization thread for the stack. It creates a new configuration, configures IP, TCP, and UDP, and then boots up the stack.

The example is from the `ndk.c` file in several of the NDK examples. It performs the following actions:

1. Create and start a timer object that will be used by the NDK by calling the POSIX *timer_create()* and *timer_settime()* functions. Internally, these POSIX functions may use any supported RTOS, such as TI-RTOS Kernel or FreeRTOS.
2. Initiate a system session by calling *NC_SystemOpen()*.
3. Create a new configuration by calling *CfgNew()*.
4. Configure the stack's settings for IP, TCP, and UDP. See the *initIp()*, *initTcp()*, and *initUdp()* functions in `ndk.c` for an example of how to configure these settings with calls to *CfgAddEntry()*.
5. Configure the stack sizes used for low, normal, and high priority tasks by calling *CfgAddEntry()*.
6. Boot the system using this configuration by calling *NC_NetStart()*.
7. Free the configuration on system shutdown (when *NC_NetStart()* returns) and call *CfgFree()* and *NC_SystemClose()*.
8. Call *timer_delete()* to delete the timer object.

```
static void *ndkStackThread(void *threadArgs)
{
    void *hCfg;
    int rc;
    timer_t ndkHeartBeat;
    struct sigevent sev;
    struct itimerspec its;
    struct itimerspec oldIts;
    int ndkHeartBeatCount = 0;

    /* create the NDK timer tick */
    sev.sigev_notify = SIGEV_SIGNAL;
    sev.sigev_value.sival_ptr = &ndkHeartBeatCount;
    sev.sigev_notify_attributes = NULL;
    sev.sigev_notify_function = &llTimerTick;

    rc = timer_create(CLOCK_MONOTONIC, &sev, &ndkHeartBeat);
```

```
if (rc != 0) {
    DbgPrintf(DBG_INFO, "ndkStackThread: failed to create timer (%d)\n");
}

/* start the NDK 100ms timer */
its.it_interval.tv_sec = 0;
its.it_interval.tv_nsec = 100000000;
its.it_value.tv_sec = 0;
its.it_value.tv_nsec = 100000000;

rc = timer_settime(ndkHeartBeat, 0, &its, NULL);
if (rc != 0) {
    DbgPrintf(DBG_INFO, "ndkStackThread: failed to set time (%d)\n");
}

rc = NC_SystemOpen(NC_PRIORITY_LOW, NC_OPMODE_INTERRUPT);
if (rc) {
    DbgPrintf(DBG_ERROR, "NC_SystemOpen Failed (%d)\n");
}

/* create and build the system configuration from scratch. */
hCfg = CfgNew();
if (!hCfg) {
    DbgPrintf(DBG_INFO, "Unable to create configuration\n");
    goto main_exit;
}

/* IP, TCP, and UDP config */
initIp(hCfg);
initTcp(hCfg);
initUdp(hCfg);

/* config low priority tasks stack size */
rc = 2048;
CfgAddEntry(hCfg, CFGTAG_OS, CFGITEM_OS_TASKSTKLOW, CFG_ADDMODE_UNIQUE,
        sizeof(uint32_t), (unsigned char *)&rc, NULL);

/* config norm priority tasks stack size */
rc = 2048;
CfgAddEntry(hCfg, CFGTAG_OS, CFGITEM_OS_TASKSTKNORM, CFG_ADDMODE_UNIQUE,
        sizeof(uint32_t), (unsigned char *)&rc, NULL);

/* config high priority tasks stack size */
rc = 2048;
CfgAddEntry(hCfg, CFGTAG_OS, CFGITEM_OS_TASKSTKHIGH, CFG_ADDMODE_UNIQUE,
        sizeof(uint32_t), (unsigned char *)&rc, NULL);
do {
    rc = NC_NetStart(hCfg, networkOpen, networkClose, networkIPAddr);
} while(rc > 0);

/* Shut down the stack */
```

```
    CfgFree(hCfg);

main_exit:
    NC_SystemClose();

    /* stop and delete the NDK heartbeat */
    its.it_value.tv_sec = 0;
    its.it_value.tv_nsec = 0;

    rc = timer_settime(ndkHeartBeat, 0, &its, &oldIts);
    rc = timer_delete(ndkHeartBeat);
    DbgPrintf(DBG_INFO, "ndkStackThread: exiting ...\n");
    return (NULL);
}
```

2.1.4.1.2 Constructing a Configuration using the DHCP Client Service

This section examines the *initIp()* function from `ndk.c` that was called in the previous section. The function tells the stack to use the Dynamic Host Configuration Protocol (DHCP) 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. 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. Add a configuration entry for the local hostname using the hostName, which is declared as `static char *hostName = "tisoc";`
2. Set the elements of dhcpc, which has a structure of type `CI_SERVICE_DHCPC`. This structure is described in the *TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524).
3. Add a configuration entry specifying the DHCP client service to be used.

```
static void initIp(void *hCfg)
{
    CI_SERVICE_DHCPC dhcpc;
    unsigned char    DHCP_OPTIONS[] = { DHCPOPT_SUBNET_MASK };

    /* Add global hostname to hCfg (to be claimed in all connected domains) */
    CfgAddEntry(hCfg, CFGTAG_SYSINFO, CFGITEM_DHCP_HOSTNAME, 0,
            strlen(hostName), (unsigned char *)hostName, NULL);

    /* Use DHCP to obtain IP address on interface 1 */

    /* Specify DHCP Service on IF specified by "IfIdx" */
    memset(&dhcpc, 0, sizeof(dhcpc));
    dhcpc.cisargs.Mode   = CIS_FLG_IFIDXVALID;
    dhcpc.cisargs.IfIdx  = 1;
    dhcpc.cisargs.pCbSrv = &serviceReport;
```

```
    dhcpc.param.pOptions = DHCP_OPTIONS;
    dhcpc.param.len = 1;
    CfgAddEntry(hCfg, CFGTAG_SERVICE, CFGITEM_SERVICE_DHCPCLIENT, 0,
            sizeof(dhcpc), (unsigned char *)&dhcpc, NULL);
}
```

2.1.4.1.3 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()*).

```
uint32_t 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), (unsigned char *)&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 Adding Status Report Services. 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( uint32_t Item, uint32_t Status, uint32_t Report, void *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)) )
    {
        uint32_t 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), (unsigned char *)&IPTmp, 0 );
    }
}
```

2.1.4.2 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(uint32_t), (unsigned char *)&rc, 0 );
```

2.1.4.3 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
uint32_t rc = DBG_NONE;

CfgAddEntry( hCfg, CFGTAG_OS, CFGITEM_OS_DBGABORTLEVEL,
    CFG_ADDMODE_UNIQUE, sizeof(uint32_t), (unsigned char *)&rc, 0 );
```

2.1.4.4 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.

2.1.4.4.1 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( void *hCfg )
{
    unsigned char *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);
}
```

2.1.4.4.2 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;
    void          *hCfg;
    unsigned char *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, networkOpen, networkClose, networkIPAddr );
    } while( rc > 0 );

    // Delete Configuration
    CfgFree( hCfg );

// Close the OS
main_exit:
    NC_SystemClose();
```

```
    return(0);
}
```

2.1.5 NDK Initialization

Before a sockets application like the example shown in 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 a networking application. An explanation of how NETCTRL works and how it can be tuned is provided in Section 3.

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.

2.1.5.1 The NETCTRL Task Thread

If you use Cfg*() API calls for configuration, you must create a Task thread that contains a call to *NC_NetStart()*, which in turn runs the network scheduler function. NSP example applications that provide this thread in their main C source file.

If you use XGCONF for configuration, this thread is automatically generated, and the code that calls *NC_NetStart()* is in the generated C file (for example, `client_p674.c` for the evmOMAPL138 client example).

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.

2.1.5.2 Pre-Initialization

If you use Cfg*() API calls for configuration, your application must call the primary initialization function *NC_SystemOpen()* before calling any other 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:

```
rc = NC_SystemOpen( NC_PRIORITY_LOW, NC_OPMODE_INTERRUPT );
if( rc )
{
    printf("NC_SystemOpen Failed (%d)\n",rc);
    for(;;);
}
```

2.1.5.3 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 use Cfg*() API calls for configuration, 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 *networkOpen()* function below opens a user SMTP server application by calling an open function to create the main application thread.

```
static SMTP_Handle hSMTP;

//
// networkOpen
//
// This function is called after the configuration has booted
//
static void networkOpen( )
{
    // 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 *networkClose()* callback function. Therefore, the *networkClose()* function must undo what was done in *networkOpen()*.

```
//
// networkClose
//
// This function is called when the network is shutting down
//
static void networkClose()
{
    // 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, uint32_t IfIdx, uint32_t 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, (unsigned char)(IPTmp>>24)&0xFF,
        (unsigned char)(IPTmp>>16)&0xFF, (unsigned char)(IPTmp>>8)&0xFF, (unsigned char)IPTmp&0xFF );
}
```

2.1.5.4 Network Startup

If you use Cfg*() API calls for configuration, 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 an 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 shut down 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, networkOpen, networkOpen, networkIPAddr );
} while( rc > 0 );
```

2.1.5.5 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 use Cfg*() API calls for configuration, 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( uint32_t Item, uint32_t Status, uint32_t Report, void *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 Constructing a Configuration using the DHCP Client Service.

2.2 Configuring the NDK with XGCONF

As an alternative to configuring NDK applications with C code that calls Cfg*() functions, you can use the XGCONF configuration tool within CCStudio. Graphical displays let you enable and set properties as needed. XGCONF may be used to configure the TI-RTOS Kernel. The same configuration in one project can configure both NDK and TI-RTOS Kernel modules and objects.

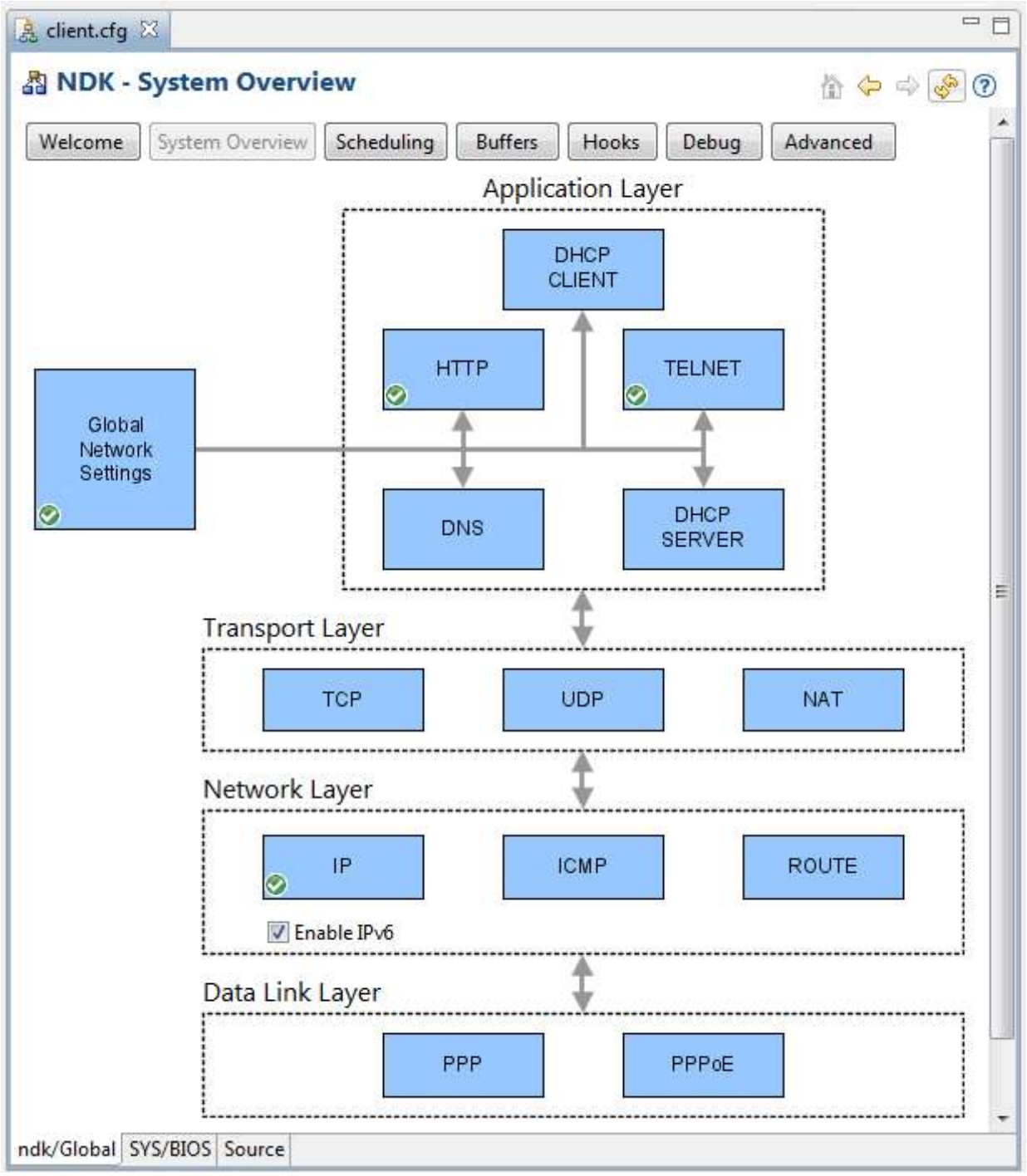
> **NOTE:** XGCONF is not supported for applications that use FreeRTOS.

When you create a project using a TI-RTOS Kernel template, the project will contain a configuration file (`*.cfg`) that can be edited with the XGCONF graphical editor in CCStudio. 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.

You can configure the NDK modules through the XGCONF configuration editor. (Internally, the same configuration database is updated when the `*.cfg` file is built.)

> **NOTE:** You must choose one method or the other to configure your application. You should *not* mix configuration methods. If you have NDK applications that use the 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 Configuring NDK Modules for how to use XGCONF with the NDK. For more details, see Chapter 2 of the *TI-RTOS Kernel (SYS/BIOS) User's Guide* (SPRUEX3).
2. When XGCONF opens, you see the Welcome sheet for TI-RTOS Kernel. You should see categories for the **NDK Core Stack** and your NSP in the Available Products area. If you do not, your CCStudio Project does not have NDK support enabled. See 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.



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.

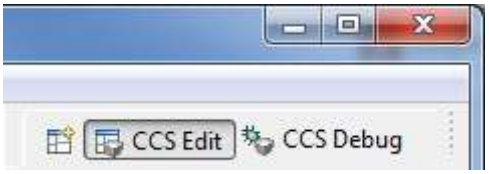
2.2.1 Opening the XGCONF Configuration Editor

When you create a project using a TI-RTOS Kernel template, the project will contain a configuration file (`*.cfg`) that can be edited with the XGCONF graphical editor in CCStudio. 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-RTOS Kernel (SYS/BIOS) User's Guide* (SPRUEX3)

To open XGCONF, follow these steps:

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



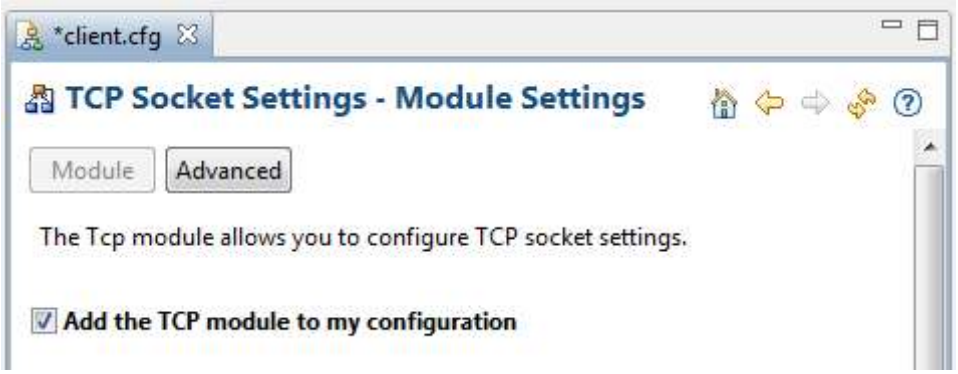
1. Double-click on the `*.cfg` configuration file in the Project Explorer tree. While XGCONF is opening, the CCStudio status bar shows that the configuration is being processed and validated.
2. When XGCONF opens, you see the Welcome sheet for the TI-RTOS Kernel. You should see categories for the **NDK Core Stack** and your NSP in the Available Products area. If you do not, your CCStudio Project does not have NDK support enabled. See 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

1. You will see the Welcome sheet for NDK configuration. This sheet provides an overview of the NDK, configuration information, and documentation for the NDK.
2. 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.

2.2.2 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** from the context menu.)

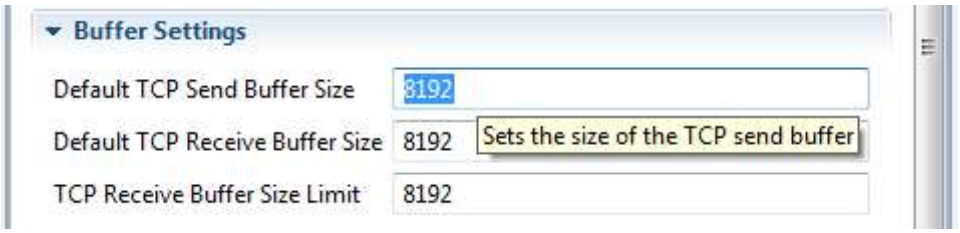


1. Notice that the module you added to the configuration is now listed in the **Outline** view.

2.2.3 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.

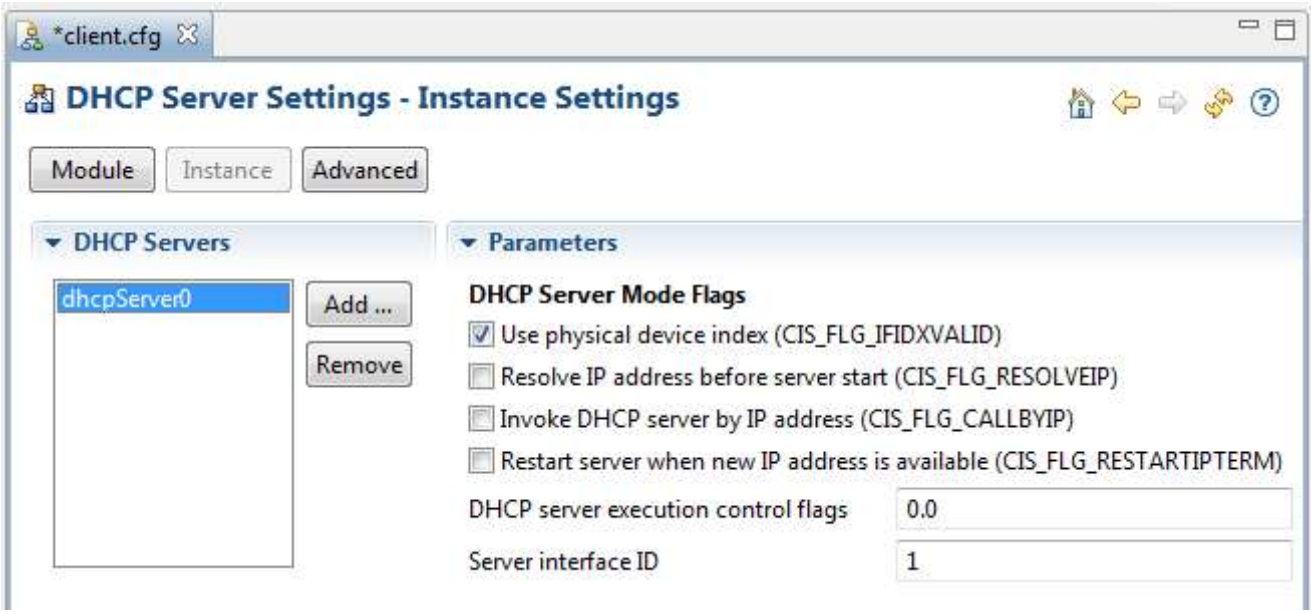


For details about properties, right-click and choose **Help** from the context menu. This opens the CDOC online reference system for the NDK. The properties names listed in this online help are the names used in the configuration source file. You can click the **Source** tab at the bottom of the XGCONF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

2.2.4 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.



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

2.2.5 Saving Changes to the Configuration

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

2.2.6 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
- 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.

2.2.7 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.

2.2.7.1 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.

2.2.7.2 Priority Levels for Network Tasks

The stack is designed to be flexible, and has an OS adaptation layer that can be adjusted to support any system software environment that is built on top of the TI-RTOS Kernel. 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 *NDK_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 *NDK_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 Section 3 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 Choosing the llEnter()/llExit() Exclusion Method, or by using the Semaphore based version of the OS library: OS_SEM.

2.2.7.2.1 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.

2.2.7.3 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.

2.2.8 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-RTOS Kernel (SYS/BIOS) 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 the TI-RTOS Kernel.

2.2.9 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 System Configuration, 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 Saving and Loading a Configuration as hook functions to save the configuration created with XGCONF and reload if from non-volatile memory on startup.

2.2.10 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.

2.2.11 Advanced Global Configuration

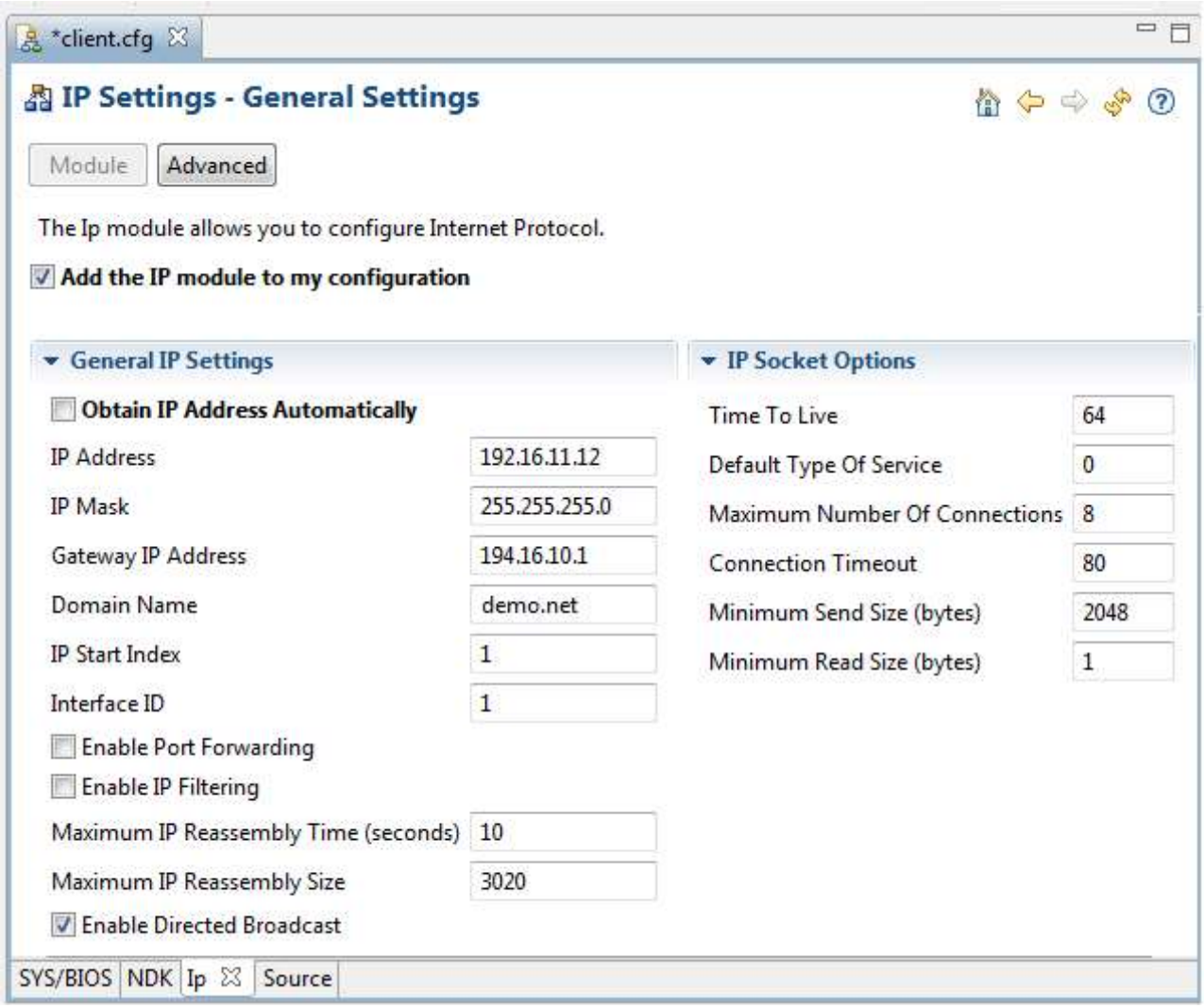
You can configure additional global NDK properties by selecting the NDK's **Global** module and then clicking the **Advanced** button. You should be careful when setting these properties. In general, it is best to leave these properties set to their defaults. Some example advanced properties are:

- *Global.ndkTickPeriod* lets you adjust the NDK heartbeat rate. The default is 100 ticks. This matches the default TI-RTOS Kernel Timer object, which drives the 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.
- *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.

2.2.12 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.



1. 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.
2. 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.

2.3 Creating a Task

Applications that use the NDK may create Task threads in either of the following ways:

- **Call TaskCreate()** as described in the *TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524).
- **Call the native thread create APIs** corresponding to the RTOS you are using. For example, SYS/BIOS users would call Task_create(). Note that if you use native thread APIs to create task threads directly, the code of your thread function needs to initializes the file descriptor table (see "Initializing the File Descriptor Table") by calling *fdOpenSession()* at the beginning, and *fdCloseSession()* at the end.

Internally, *TaskCreate()* calls *fdOpenSession()* and *fdCloseSession()* automatically, and then calls the native OS API (corresponding to the RTOS the app is using) to create a thread

Priority-based exclusion makes it important that your application use only a priority in the range of the configured NDK priorities (`OS_TASKPRILOW` to `OS_TASKPRIHIGH`). 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.

The following example uses *TaskCreate()* to create a Task with a normal priority level:

```
void *taskHandle = NULL;

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

The following example uses SYS/BIOS APIs to create a Task for use by the NDK. The thread has a normal priority level and a stack size of 2048.

```
#include <ti/sysbios/BIOS.h>
#include <ti/sysbios/knl/Task.h>
#include <ti/ndk/inc/netmain.h>

void myNetThreadFxn()
{
    fdOpenSession((void *)Task_self());

    /* do socket calls */

    fdCloseSession((void *)Task_self());
}

void createNdkThread()
{
    int status;
    Task_Params params;
    Task_Handle myNetThread;

    Task_Params_init(&params);
    params.instance->name = "myNetThread";
    params.priority = OS_TASKPRINORM;
    params.stackSize = 2048;

    myNetThread = Task_create((Task_FuncPtr)myNetThreadFxn, &params, NULL);

    if (!myNetThread) {
        /* Error */
    }
}
```

2.3.1 Initializing the File Descriptor Table

Each Task thread that uses the sockets or file API must allocate a file descriptor table and associate the table with the Task handle. The following code shows how to do this using SYS/BIOS Task APIs. This process is

described fully in the *TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524). To accomplish this, a call to *fdOpenSession()* must be performed before any file descriptor oriented functions are used, and then *fdCloseSession()* should be called when these functions are no longer required.

```
void mySocketsFunction()
{
    fdOpenSession((void *)Task_self());

    /* do socket calls */

    fdCloseSession((void *)Task_self());

    return (NULL);
}
```

2.4 Application Debug and Troubleshooting

Although there is 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.

2.4.1 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 "Priority Levels for Network Tasks". It is also recommended that application programmers fully understand the workings of the NETCTRL module. For this, see Section 3.

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 configuration that is calling the driver function *llTimerTick()* every 100 ms.

Not receiving any packets - ever

- When polling for data by making *NDK_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 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 NDK_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 "Packet Buffer Pool").
- Verify you do not have any scheduling issues. Try running the scheduler in high priority (via *NC_SystemOpen()*).

UDP application drops packets on NDK_recv() calls.

- Make sure you have plenty of packet buffers available (see "Packet Buffer Pool")).
- Make sure the packet threshold for UDP is high enough to hold all UDP data received in between calls to *NDK_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 the `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 OS Adaptation 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 use Cfg*() API calls for configuration, you can configure these IP module properties by using the following C code:

```
uint32_t tmp = 65500;

// configure NDK
CfgAddEntry(hCfg, CFGTAG_IP, CFGITEM_IP_IPREASMMAXSIZE,
            CFG_ADDMODE_UNIQUE, sizeof(uint32_t), (unsigned char*) &tmp, 0);
CfgAddEntry(hCfg, CFGTAG_IP, CFGITEM_IP_SOCKUDPRXLIMIT,
            CFG_ADDMODE_UNIQUE, sizeof(uint32_t), (unsigned char*) &tmp, 0);

// set socket options
NDK_setsockopt(s, SOL_SOCKET, SO_RCVBUF, &tmp, sizeof(int) );
NDK_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)
- OS Adaptation Layer Definitions:
  - Change the `MMALLOC_MAXSIZE` definition in the `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 use Cfg*() API calls 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 OS Adaptation Layer library in `/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 *NDK_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(unsigned char *pIpHdr) {
    ...
}

NDK_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(unsigned char *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.)

2.4.2 Debug Messages

Debug messages for TI-RTOS Kernel are handled using the *System_printf()* API, which is provided by XDCtools for use by TI-RTOS.

Debug output for FreeRTOS is not currently supported, so the subsections that follow do not apply to applications that use FreeRTOS. You may modify the OS Adaptation Layer source code to add other types of program output as desired.

2.4.2.1 Controlling Debug Messages

Debug messages for TI-RTOS Kernel 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 shut down.

By default, all debug messages are printed, and messages with a level of DBG_ERROR causes a stack shutdown. This behavior can be modified in the configuration as described in "Global Debug Configuration". Or, you can modify it through the system configuration as described in "Controlling NDK and OS Options via the Configuration" and "Shutdown". Also see the *TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524).

2.4.2.2 Interpreting Debug Messages

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

2.4.2.2.1 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.

2.4.2.2.2 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.

2.4.2.2.3 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 "Memory Corruption" for possible causes.

2.4.2.2.4 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.

2.4.2.2.5 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 "Memory Corruption" for possible causes.

2.4.2.2.6 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.

2.4.3 Memory Corruption

Memory corruption errors may occur as 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.

2.4.4 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.

2.4.5 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 where 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.

2.4.5.1 mmCheck - Generate Memory Manager Report

Syntax

```
void _mmCheck( uint32_t CallMode, int (*pPrn)(const char *,...) );
```

Parameter	Description
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 uint32_t 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

3 Network Control Functions

This chapter describes the network control functions.

3.1 Introduction to NETCTRL Source

3.1.1 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.

3.1.2 NETCTRL Source Files

Source code to the NETCTRL library consists of two C files located in the `/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

3.1.3 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.

3.1.4 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 *networkOpen()* 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 *networkOpen()* 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()*.

3.1.5 Booting and Scheduling

"NDK Initialization" 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.

3.2 NETCTRL Scheduler

3.2.1 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 "Choosing the llEnter()/llExit() Exclusion Method". 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.

3.2.2 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, an 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 an 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 "Pre-Initialization" and in the *NDK Programmer's Reference Guide*.

3.2.3 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 "Pre-Initialization" 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 *NDK_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.

3.2.4 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 {
    void       *hSemEvent;
    uint32_t   EventCodes[STKEVENT_NUMEVENTS];
} STKEVENT;

#define STKEVENT_NUMEVENTS    5

#define STKEVENT_TIMER        0
#define STKEVENT_ETHERNET     1
#define STKEVENT_SERIAL       2
#define STKEVENT_LINKUP       3
#define STKEVENT_LINKDOWN     4
```

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.

You can provide a hook function to run when a driver signals a STKEVENT_LINKUP or STKEVENT_LINKDOWN event, meaning that the link has come up or gone down. Note that such a hook function will only be called if the driver has code to call STKEVENT_signal({STKEVENT_LINKUP}) and STKEVENT_signal({STKEVENT_LINKDOWN}).

The hook function should accept a single int status parameter. If the function receives 0, the link is now down (for example, because a cable was disconnected). If the function receives a 1, the link is now up. To register your hook function, call *NC_setLinkHook()* as follows:

```
NC_setLinkHook( void (*LinkHook)(int) );
```

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*.

3.2.5 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 "Scheduler Overview", 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
#define FLAG_EVENT_LINKUP    8
#define FLAG_EVENT_LINKDOWN  16

static void NetScheduler( uint32_t const SerialCnt, uint32_t 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 );
        }

        /* Clear our event flags */
        fEvents = 0;

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

        /* Check for a timer event and flag it. EventCodes[STKEVENT_TIMER] */
        /* is set as a result of llTimerTick() (NDK heartbeat) */
        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 )
{
    NIMUPacketServiceCheck (fEvents);

    /* 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. */

/* 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;
}

/* 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;
}

/* Check if link went up */
if(stkEvent.EventCodes[STKEVENT_LINKUP])
```

```
{
    /* Clear the event and record it in our flags */
    stkEvent.EventCodes[STKEVENT_LINKUP] = 0;
    fEvents |= FLAG_EVENT_LINKUP;
}

/* Check if link went down */
if(stkEvent.EventCodes[STKEVENT_LINKDOWN])
{
    /* Clear the event and record it in our flags */
    stkEvent.EventCodes[STKEVENT_LINKDOWN] = 0;
    fEvents |= FLAG_EVENT_LINKDOWN;
}

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

    /* Check for timer event. Timer event flag is set as a result of */
    /* llTimerTick() (NDK heartbeat) */
    if( fEvents & FLAG_EVENT_TIMER )
        ExecTimer();

    /* Check for packet event */
    if( fEvents & FLAG_EVENT_ETHERNET )
        NIMUPacketService();

    /* Check for serial port event */
    if( fEvents & FLAG_EVENT_SERIAL )
        llSerialService();

    /* Exit Kernel Mode */
    llExit();

    /* Check for a change in link status.  Do this outside of above */
    /* llEnter/llExit pair as to avoid illegal reentrant calls to */
    /* kernel mode by user's callback. */

    /* Check for link up status */
    if( fEvents & FLAG_EVENT_LINKUP )
    {
        /* Call the link status callback, if user registered one */
        if (NetLinkHook) {
            /* Pass callback function a link status of "up" */
            (*NetLinkHook)(1);
        }
    }

    /* Check for link down status */
```

```
            if( fEvents & FLAG_EVENT_LINKDOWN )
            {
                /* Call the link status callback, if user registered one */
                if (NetLinkHook) {
                    /* Pass callback function a link status of "down" */
                    (*NetLinkHook)(0);
                }
            }
        }
    }
}
```

3.3 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 CCStudio 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 `/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.

4 OS Adaptation Layer

The OS adaptation layer controls how the NDK uses RTOS resources. This includes its use of tasks, semaphores, and memory. In most cases, the only API described in this chapter that your NDK application will call is *TaskCreate()*.

4.1 About OS Adaptation

The NDK uses an OS adaptation layer so that applications can use any supported RTOS. Currently, the TI-RTOS Kernel and FreeRTOS are supported for NDK applications. This portability is achieved through OS adaptation APIs that are converted into native OS thread calls.

This chapter describes the APIs that perform OS adaptation.

In most cases, your user application code will call only *TaskCreate()*. However, you can modify the source code of the adaptation layer as needed. The source code for the OS adaptation layer is provided in the `/ti/ndk/os` directory:

4.1.1 Source Files

Source code for the OS library consists of the following files, which are located in the `/ti/ndk/os` directory:

File	Description
efs.c	Embedded (RAM based) file system
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

Two additional include files are located in the `/ti/ndk/inc/os` directory:

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

4.2 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 for the stack's exclusion method functions: *llEnter()* and *llExit()*. The latter are discussed in "Choosing the llEnter()/llExit() Exclusion Method" of this document.

In most cases, your user application code will call only *TaskCreate()*.

Most of the Task and Semaphore functions defined in the *TI Network Developer's Kit (NDK) API Reference Guide* (SPRU524) are macros that call an RTOS. The macros are defined in `inc/os/osif.h`. The functions that may need special handling are described in the subsections that follow.

4.2.1 TaskCreate(), TaskExit(), and TaskDestroy()

The create, exit and destroy functions all call their native OS thread API equivalents.

If you are using the TI-RTOS Kernel, for *TaskExit()* and *TaskDestroy()* to function as expected, the Task.deleteTerminatedTasks configuration parameter must be set to "true". This parameter setting instructs the Task module to delete completed Tasks in the 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 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.

4.2.2 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 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 one of the NDK defined task priorities. If you use a priority greater than the NDK's highest defined priority level (OS_TASKPRIHIGH), 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.

4.3 Packet 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.

4.3.1 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.

4.3.2 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.

4.3.3 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()*.

4.4 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. 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 *malloc()* and *free()*. 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.

4.5 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.

4.6 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.

4.7 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 RTOS 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 RTOS APIs or dynamic memory allocation method for its 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 may all be customized.
- 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 `/ti/ndk/stack/pbm` directory.

Related Documentation From Texas Instruments

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

For information about the design of the Ethernet and Serial driver architecture in the NDK, see the *Network Developer's Kit (NDK) Support Package Ethernet Driver Design Guide* (SPRUFP2).

Information about the POSIX pthread support provided by the SimpleLink SDKs and used by the NDK can be found in the POSIX Thread (pthread) Support page in the TI processors wiki.

Information about the TI-RTOS Kernel, which may be used with NDK applications, can be found in the *TI-RTOS Kernel (SYS/BIOS) User's Guide* (SPRUEX3) 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.

- 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

Trademarks

SimpleLink and Cortex are trademarks of Texas Instruments.

ARM is a registered trademark of Texas Instruments.

Windows is a registered trademark of Microsoft.
