TMS320C6000 Network Developer's Kit (NDK) Software Programmer's

Reference Guide

Literature Number: SPRU524C
January 2007

Texas Instruments

Contents

List of Figures

Read This First

About This Manual

This programmer's reference guide describes the various API functions provided by the NDK libraries, and is intended to aid the development of network applications. It is the central reference document used when programming the stack. See the *TMS320C6000 Network Developer's Kit (NDK) Software User's Guide* (SPRU523) to familiarize yourself with the stack libraries and in using the stack with DSP/BIOS™ and Code Composer Studio™ (CCStudio) Development Tools.

How to Use This Manual

This document contains the following chapters:

- **Chapter 1 - Introduction**, summarizes the various API sets described in the NDK documentation.
- **Chapter 2 - Operating System Abstraction API**, describes the API used by the adaptation layer to access the operating system.
- **Chapter 3 - Sockets and Stream IO API**, describes the file and sockets API functions.
- **Chapter 4 - Initialization and Configuration**, describes the NDK initialization and configuration, including the Configuration Manager API and the Network Control module.
- **Chapter 5 - Network Tools Library - Support Functions**, describes the network support functions contained in the NETTOOLS library.
- **Chapter 6 - Network Tools Library - Services**, describes the network servers and services contained in the NETTOOLS library.
- **Appendix A - Internal Stack Functions**, contains a partial list of internal stack functions provided to aid in the comprehension of kernel oriented calls.
- **Appendix B - Network Address Translation**, describes the optional Network Address Translation component, how to set up virtual networks, and protocol proxies.
- **Appendix C - Point-to-Point Protocol**, describes the operation of the PPP and PPPoE support API included in the NDK, and how to interface to a serial device.
- **Appendix D - Hardware Adaptation Layer (HAL)**, describes the operation of the HAL, and the HAL API functions.
- **Appendix E - Web Programming with the HTTP Server**, describes how to get information from an embedded network device through the webserver.

Notational Conventions

This document uses the following conventions:

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

Related Documentation From Texas Instruments

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

SPRU189 —***TMS320C6000 DSP CPU and Instruction Set Reference Guide.*** Describes the CPU architecture, pipeline, instruction set, and interrupts for the TMS320C6000 ™digital signal processors (DSPs).

SPRU190 —***TMS320C6000 DSP Peripherals Overview Reference Guide.*** Provides an overview and briefly describes the peripherals available on the TMS320C6000™ family of digital signal processors (DSPs).

SPRU197 —***TMS320C6000 Technical Brief.*** Provides an introduction to the TMS320C62x™ and TMS320C67x™ digital signal processors (DSPs) of the TMS320C6000™ DSP family. Describes the CPU architecture, peripherals, development tools and third-party support for the C62x™ and C67x™ DSPs.

SPRU198 —***TMS320C6000 Programmer's Guide.*** Reference for programming the TMS320C6000™ digital signal processors (DSPs). Before you use this manual, you should install your code generation and debugging tools. Includes a brief description of the C6000 DSP architecture and code development flow, includes C code examples and discusses optimization methods for the C code, describes the structure of assembly code and includes examples and discusses optimizations for the assembly code, and describes programming considerations for the C64x™ DSP.

SPRU509 —***TMS320C6000 Code Composer Studio Development Tools v3.3 Getting Started Guide***. Introduces some of the basic features and functionalities in Code Composer Studio™ to enable you to create and build simple projects.

SPRU523 —***TMS320C6000 Network Developer's Kit (NDK) Software User's Guide.*** Describes how to use the NDK libraries, how to develop networking applications on TMS320C6000™ platforms, and ways to tune the NDK to fit a particular software environment.

Trademarks

DSP/BIOS, Code Composer Studio, TMS320C6x, TMS320C6000, TMS320C62x, TMS320C67x, C62x, C67x, C64x are trademarks of Texas Instruments.

Introduction

This chapter serves as an introduction to the programming API reference for the TMS320C6000™ NDK Software.

Topic **Page**

1.1 What This Document Covers

This Programmer's Reference Guide for the NDK is mainly a programming API reference guide. It is intended to aid in the development of network applications and describes the various API functions provided by the stack libraries.

Although this Programmer's Reference Guide will be the central reference document used when programming the stack, you should first see the *TMS320C6000 Network Developer's Kit (NDK) Software User's Guide* (SPRU523) to familiarize yourself with the stack libraries, and in using the stack with DSP/BIOS™ and the Code Composer Studio™ (CCStudio) Development Tools.

1.1.1 Supplemental API Information

The following information appears as appendices to this document. These sections contain optional information that may be useful in understanding the low-level application interface, but is not required when developing traditional network applications.

- Appendix A *Internal Stack Functions*

  The stack library internal function specification describes a subset of the low-level programming interface to the stack. These functions allow the application writer to make use of kernel level function APIs. As a general rule, it is not necessary to use this API for application development, although some of the sample applications included in the NDK make use of these function calls.
- Appendix B *Network Address Translation (NAT)*

  The stack library includes Network Address Translation module. This appendix describes the operational theory of NAT, and how to use the NAT functions included in the library.
- Appendix C *Point-to-Point Protocol (PPP)*

  The stack library has internal device sections for both traditional Ethernet, and PPP. The PPP module can act as PPP client, server, or both (assuming multiple interfaces). This appendix describes the operation of the PPP module, the PPP over Ethernet (PPPoE) module, and how to interface an HDLC based serial device.
- Appendix D *Hardware Adaptation Layer (HAL)*

  Appendix D describes the hardware and operating system interfaces used by the stack. The information allows application programmers to call device drivers directly when needed. This appendix does not supply information about porting the HAL to a new platform.
- Appendix E *Web Programming with the HTTP Server*

  Appendix E describes how to make use of the HTTP server included in the NDK. The main topics covered are adding Web content and writing CGI functions. There is also a description of the HTTP API used by CGI functions, and some CGI example applications.

Operating System Abstraction API

To keep the stack system portable, it was coded to a very compact operating system abstraction. The stack can execute in any operating environment by porting the functions described here. Most of these functions will map directly to their native OS counterpart.

If you program to this API, your applications will execute on any system to which this abstraction is ported, but more importantly, because all the NDK functions are written to this layer, the behavior of the NDK can be altered by altering the implementation of this layer. This allows the stack to be tuned in how it interfaces to the native operating system.

Topic **Page**

2.1 Operating System Configuration

2.1.1 Synopsis

The OS has a couple of configuration options that regulate its behavior. These are stored in a data structure. The types of properties defined in the structure are those that would typically be macros, but using a data structure allows the values to be changed without rebuilding the libraries.

The structure is described here for completeness, but applications should use the configuration system to make alterations to these values. The configuration system is described later in this document.

2.1.2 Configuration Structure

The stack internal configuration structure is _**oscfg**. Any element in this structure may be modified before the system is booted. System initialization is covered later in this document.

The _oscfg structure is of type OSENVCFG, which is defined as follows:

```
// Configuration Structure
typedef struct _osenvcfg {
    uint     DbgPrintLevel;     // Debug message print threshold
    uint     DbgAbortLevel;     // Debug message sys abort threshold
    int      TaskPriLow;        // Lowest priority for stack task
    int      TaskPriNorm;       // Normal priority for stack task
    int      TaskPriHigh;       // High priority for stack task
    int      TaskPriKern;       // Kernel-level priority (highest)
    int      TaskStkLow;        // Minimum stack size
    int      TaskStkNorm;       // Normal stack size
    int      TaskStkHigh;       // Stack size for high volume tasks
} OSENVCFG;
```

The structure entries as defined as follows:

_oscfg.DbgPrintLevel *Debug message print threshold*

Default Value DBG_INFO

Description This is the lowest severity level of a system debug message (call to *DbgPrintf()* function) that will be recorded into the debug log. The threshold may be raised. The legal values for this variable are: DBG_INFO, DBG_WARN, DBG_ERROR, and DBG_None.

_oscfg.DbgAbortLevel *Debug message abort threshold*

Default Value DBG_ERROR

Description This is the lowest severity level of a system debug message (call to *DbgPrintf()* function) that will result in a system shutdown (call to *NC_NetSop()*). The threshold may be raised. The legal values for this variable are: DBG_INFO, DBG_WARN, DBG_ERROR, and DBG_None.

_oscfg.TaskPriLow *Priority Level for Low Priority Stack Task*

Default Value 3

Description This is the priority at which low priority stack task threads are set. Setting a thread to a lower priority than this will not disrupt the system, but no system or service supplied in this package will attempt it.

_oscfg.TaskPriNorm *Priority Level for Normal Priority for Stack Task*

Default Value 5

Description This is the priority at which most stack task threads are set. Task threads that are created by the system or services will usually run at this level.

_oscfg.TaskPriHigh *Priority Level for High Priority for Stack Task*

Default Value 7

Description This is the priority at which high priority stack task threads are set. Setting a thread at a higher priority than this may disrupt the system and cause unpredictable behavior if the thread calls any stack related functions. High priority tasks (like interrupts) can execute at higher priority levels, but should signal lower priority tasks to perform any required stack functions.

_oscfg.TaskPriKern *Priority Level of High Priority Kernel Tasks*

Default Value 9

Description This is the priority that task threads execute at when they are inside the kernel. Setting tasks to this priority level ensures that they will not be disrupted by another task calling stack functions. Note that this priority should be 2 higher than _oscfg.TaskPriHigh, to allow the scheduler thread to occupy a priority in between. The proper method of entering the kernel is to call *llEnter()* and *llExit()*. These functions are discussed in the appendices, as they are not required for applications programming.

_oscfg.TaskStkLow *Minimum Task Stack Size*

Default Value 3072

Description This is the stack size used for network task that do very little network processing, or do not use TCP.

_oscfg.TaskStkNorm *Normal Task Stack Size*

Default Value 4096

Description This is the stack size used for a network task with an average network bandwidth using TCP. It is used for the majority of network tasks in the network tools library that use TCP.

_oscfg.TaskStkHigh *High Volume Task Stack Size*

Default Value 5120

Description This is the stack size used to network tasks that require a high network bandwidth using TCP. It is also used for tasks calling HTTP CGI functions.

2.2 Task Support

2.2.1 Synopsis

The task object provides a method of manipulating task threads using a generic task handle. Task threads are executed on a priority based method, with a least-recently-run algorithm used on those with equal priority. Each task thread has its own private stack.

DSP/BIOS Users Note: Task handles created and used by this abstraction are compatible and interchangeable with DSP/BIOS TSK handles.

2.2.2 Function Overview

The Task Object access functions (in functional order) are as follows:

TaskCreate()	Create new task thread
TaskDestroy()	Destroy a task thread
TaskSelf()	Get handle to current task thread
TaskExit()	Exit (terminate) current task thread
TaskYield()	Yield to another task thread at the same priority
TaskSleep()	Block a task thread for a period of time
TaskBlock()	Block a task thread
TaskSetPri()	Set task thread priority level
TaskGetPri()	Get task thread priority level
TaskSetEnv()	Assign one of three private environment handles to task thread
TaskGetEnv()	Retrieve one of three private environment handles

2.2.3 Task API Functions

TaskBlock — ***Block Task From Execution***

Syntax

void TaskBlock(HANDLE hTask);

Parameters

hTask	Handle to target task

Return Value

None.

Description

Permanently blocks the specified task from execution.

Calling this function may cause a task switch.

TaskCreate — *Create a Task Thread*

Syntax

HANDLE TaskCreate(void(*pFun)(), char *Name, int Priority, uint StackSize, UINT32 Arg1, UINT32 Arg2, UINT32 Arg3);

Parameters

pFun	Pointer to task entry-point function
Name	NULL terminated task name (truncated after 11 characters)
Priority	Task priority level (0-15)
StackSize	Task stack size
Arg1	Optional task function argument 1
Arg2	Optional task function argument 2
Arg3	Optional task function argument 3

Return Value

Returns a Task Handle on success or NULL on memory failure.

Description

Creates a new task object. If successful, *TaskCreate()* returns a handle to the newly created task.

The task name supplied in *Name* is used for informational purposes only, and does not need to be unique.

The task priority specified in *Priority* determines the task thread's priority relative to other tasks in the system. The value of Priority is constrained only by the size of an int on the target environment, but a range of 0 to 15 is recommended. 0 is the lowest priority and should be reserved for an idle task. If the specified priority is negative, the task is blocked.

The task stack size specified by *StackSize* is not examined or adjusted by the create function. The size should be made compatible with the native environment (a multiple of 4 bytes should be sufficient).

Arg1 through Arg3 are optional arguments that can be passed to the calling function (they are always pushed onto the stack, but the task function need not reference them).

There is no limit to the number of tasks that can be installed in the system. The only possible failure on *TaskCreate()* is a memory allocation error.

If the priority level of the new task is higher than the priority level of the current task, the entry-point function pFun is executed immediately (before *TaskCreate()* returns to the caller).

Calling this function may cause a task switch.

TaskDestroy — *Destroy a Task Thread*

Syntax

void TaskDestroy(HANDLE hTask);

Parameters

hTask	Handle to target task

Return Value

None.

Description

Terminates execution of the task object specified by the supplied handle *hTask*, and frees task object from system memory. Note that memory allocated by the task thread is not associated with the task thread and must be freed manually.

TaskExit — *Exit a Task Thread*

Syntax

void TaskExit();

Parameters

None.

Return Value

Does not return.

Description

This function exits a task thread. It should always be called immediately before the task entry-point function is about to return, but it may be called from anywhere.

TaskGetEnv — *Get Task Environment Handle*

Syntax

HANDLE TaskGetEnv(HANDLE hTask, int Slot);

Parameters

hTask	Handle to target task
Slot	Environment slot to use (1-3)

Return Value

Private environment handle or NULL.

Description

Returns a private environment handle for the supplied task handle hTask that was previously stored with the *TaskSetEnv()* function. The slot specified in *Slot* specifies the address (1-3) of the environment handle. There are actually four slots, but slot 0 is reserved.

DSP/BIOS Users Note: The OS adaptation layer (OS.LIB) implements this function for slot 0 only. The reserved slot 0 is the only slot required by the NDK. Slots 1 to 3 are not implemented. You should use the standard DSP/BIOS function *TSK_setEnv()* and *TSK_getEnv()* for private environment pointer storage and retrieval.

TaskGetPri — *Get Task Priority*

Syntax

int TaskGetPri(HANDLE hTask);

Parameters

hTask	Handle to target task

Return Value

Task priority level.

Description

Returns the priority of the target task. See *TaskSetPri()* for more information on priority.

TaskSelf — *Get the Handle to the Currently Executing Task Thread*

Syntax

HANDLE TaskSelf();

Parameters

None.

Return Value

Handle to currently executing thread, or NULL on error.

Description

Returns the task handle of the currently executing task thread. This function is used mainly in other task object calls where the caller wishes to operate on the current thread, but does not know the current thread's handle.

If called on an illegal (system) thread, this function returns NULL. Only certain implementations of the OS even have a system thread, and no user code should ever be executed on it. A NULL may also result if Task functions are called before the operating system is initialized.

TaskSetEnv — *Set Task Environment Handle*

Syntax

void TaskSetEnv(HANDLE hTask, int Slot, HANDLE hEnv);

Parameters

hTask	Handle to target task
Slot	Environment slot to use (1-3)
hEnv	Private environment handle

Return Value

None.

Description

Sets and stores a private environment handle for the supplied task handle *hTask*. This handle can be later retrieved by *TaskGetEnv()*. The slot specified in Slot assigns an address (1-3) to the environment handle. There are actually four slots, but slot 0 is reserved.

DSP/BIOS Users Note: The OS adaptation layer (OS.LIB) implements this function for slot 0 only. The reserved slot 0 is the only slot required by the NDK. Slots 1 to 3 are not implemented. Application programmers should use the standard DSP/BIOS function *TSK_setEnv()* and *TSK_getEnv()* for private environment pointer storage and retrieval.

TaskSetPri — *Set Task Priority*

Syntax

int TaskSetPri(HANDLE hTask, int Priority);

Parameters

hTask	Handle to target task
Priority	Task priority level

Return Value

Previous task priority level.

Description

Sets the priority of the target task to the specified value. The value of *Priority* is constrained only by the size of an *int* on the target environment, but a range of 0 to 15 is recommended. 0 is the lowest priority and should be reserved for an idle task. If the specified priority is negative, the task is blocked.

Calling this function may cause a task switch.

TaskSleep — *Sleep Task for Period of Time*

Syntax

void TaskSleep(UINT32 Delay);

Parameters

Delay	Time (in milliseconds) of sleep

Return Value

None.

Description

Sleeps the calling task for a period of time as supplied in *Delay*. The sleep time cannot be zero.

Calling this function may cause a task switch.

TaskYield — *Yield Execution to Another Task Thread*

Syntax

void TaskYield();

Parameters

None.

Return Value

None.

Description

This function yields execution to another thread by causing a round-robin task switch among ready task threads executing at the same priority level.

This function always causes a task switch; however, the original calling task may be the next to execute.

2.3 Semaphore Support

2.3.1 Synopsis

The semaphore object provides a method of manipulating counting semaphores using a generic handle. Semaphores can be used for both task synchronization and mutual exclusion.

DSP/BIOS Users Note: Task handles created and used by this abstraction are compatible and interchangeable with DSP/BIOS SEM handles.

2.3.2 Function Overview

The Semaphore Object access functions (in functional order) are as follows:

SemCreate()	Create new semaphore
SemDelete()	Delete semaphore
SemPend()	Wait on semaphore, optionally for a period of time
SemCount()	Get the current semaphore count
SemPost()	Release semaphore - increment count
SemReset()	Reset semaphore and set new count

2.3.3 Semaphore API Functions

SemCreate — ***Create New Semaphore***

Syntax

HANDLE SemCreate(int Count);

Parameters

Count	Initial semaphore count

Return Value

Handle to semaphore or NULL on error.

Description

Creates a new semaphore object with an initial count.

SemCount — *Get Current Semaphore Count*

Syntax

int SemCount(HANDLE hSem);

Parameters

hSem	Handle to Semaphore

Return Value

Current semaphore count

Description

Returns the current count of the semaphore object.

SemDelete — *Delete Semaphore*

Syntax

void SemDelete(HANDLE hSem);

Parameters

hSem	Handle to Semaphore

Return Value

None.

Description

Deletes the semaphore object and frees related memory.

Any task currently waiting on this semaphore is blocked forever - even if it originally specified a timeout to *SemPend()*. With a little care in programming, this will not occur.

SemPend — *Wait for a Semaphore*

Syntax

int SemPend(HANDLE hSem, UINT32 Timeout);

Parameters

hSem	Handle to Semaphore
Timeout	Maximum time to wait (in milliseconds)

Return Value

The function returns 1 if the semaphore was obtained, and 0 if not.

Description

This function waits on a semaphore.

If the semaphore count is greater than 0, the semaphore count is decrement and this function immediately returns.

If the semaphore count is zero, the task is placed on a waiting list for the semaphore and blocked. If the semaphore becomes available in the time period specified in *Timeout*, the function returns. However, the function returns regardless once the timeout has expired. A timeout value of 0 always returns without blocking or yielding. A timeout value of SEM_FOREVER causes the caller to wait on the semaphore without time out.

The waiting list is first in, first out, without regard to priority. Thus, semaphores can be used to round-robin task threads at different priority levels.

Calling this function may cause a task switch (unless called with *Timeout* set to 0).

SemPost — *Signal a Semaphore*

Syntax

void SemPost(HANDLE hSem);

Parameters

hSem	Handle to Semaphore

Return Value

None.

Description

If the semaphore count is greater than 0 (or is equal to 0, but without any pending task threads), the semaphore count is incremented and this function immediately returns.

If the semaphore count is zero and there are tasks threads pending on it, the count remains at zero, and the first thread in the pending list is unblocked.

Calling this function may cause a task switch.

SemReset — *Reset Semaphore*

Syntax

void SemReset(HANDLE hSem, int Count);

Parameters

hSem	Handle to Semaphore
Count	Initial semaphore count

Return Value

None.

Description

This function resets the semaphore, first setting an initial semaphore count, and then unblocking all tasks that are pending on the semaphore.

This function should be used with care. Tasks that are pending on the semaphore may exhibit unexpected behavior because all tasks pending on the semaphore will return from their respective *SemPend()* calls regardless of requested timeout. The return value for the respective *SemPend()* calls will always be correct because one or more tasks may get the semaphore (depending on the value of *Count*), but tasks that called *SemPend()* without a timeout may assume they have obtained the semaphore without checking the *SemPend()* return value.

Calling this function may cause a task switch.

2.4 Memory Allocation Support

2.4.1 Synopsis

As part of normal stack operation, memory will be allocated and freed on a regular basis. It is therefore recommended that a memory support system have the ability to allocate and free small memory blocks in a variety of sizes, without memory fragmentation. The functions described here work on a memory bucket system of predefined fixed sizes. Although it allocates more memory than requested, when the memory is released, it can be reused without fragmentation.

2.4.2 Function Overview

The Memory Allocation access functions (in functional order) are as follows:

mmAlloc()	Allocate Small Memory Block
mmFree()	Free *mmAlloc()* Memory Block
mmBulkAlloc()	Allocate Unrestricted Memory Block
mmBulkFree()	Free *mmBulkAlloc()* Memory Block
mmCopy()	Copy a Memory Block
mmZeroInit()	Initialize a Memory Block to Zero

2.4.3 Memory Allocation API Functions

mmAlloc — *Allocate Memory Block*

Syntax void *mmAlloc(uint size);

Return Value Pointer to allocated memory or NULL on error.

Description Allocates a memory block of at least *size* bytes in length. The function should return a pointer to the new memory block, or NULL if memory is not available. **The size of the allocation cannot be more than 3068 bytes.**

mmFree — *Free Memory Block*

Syntax int mmFree(void *pv);

Return Value If a memory tracking error occurs, this function returns 0; otherwise, it returns 1.

Description Frees a previously allocated memory block by supplying the pointer that *mmAlloc()* originally returned.

mmBulkAlloc — *Allocate Bulk Memory Block*

Syntax void *mmBulkAlloc(INT32 Size);

Return Value Pointer to allocated memory or NULL on error.

Description Allocates a memory block of at least *size* bytes in length. The function returns a pointer to the new memory block, or NULL if memory is not available. The size of the allocation is not restricted.

mmBulkFree — *Free Bulk Memory Block*

Syntax void mmBulkFree(void *pv);

Return Value None.

Description Frees a previously allocated memory block by supplying the pointer that *mmBulkAlloc()* originally returned.

mmCopy — *Copy Memory*

Syntax void mmCopy(void *pDst, void *pSrc, uint size);

Return Value None.

Description Called to copy *size* bytes of data memory from the data buffer *pSrc* to the data buffer *pDst*.

mmZeroInit — *Zero Memory*

Syntax void mmZeroInit(void *pDst, uint size);

Return Value None.

Description Called to initialize *size* bytes of data memory in the data buffer *pDst* to NULL.

2.5 Print and Debug Support

2.5.1 Synopsis

The OS abstraction includes a family of compact *printf()* functions that print using a fixed buffer. The size of the buffer (max *printf()* length) is defined in the OS abstraction layer. The code to print to the standard output device is also provided, and this function can be modified to print or log as required.

The stack also provides another form of the printf function called *DbgPrintf()*. This function prints debug messages to a global debug log. The severity threshold at which the debug message is recorded can be adjusted, as well as at what point the error causes a system shutdown.

DSP/BIOS Users Note: Under DSP/BIOS, there is a minor incompatibility between the compact *printf()* function provided here and the one supplied in the RTS library. Other than not supporting floating point, this version of *printf()* treats long values (e.g., %ld) as 32 bit quantities, not 40 bits. Thus, when using DSP/BIOS, it is best to avoid the use of %ld.

2.5.2 Standard API Functions

The standard set of printf functions is supported:

```
int   printf(const char *format, ...);
int   sprintf(char *s, const char *format, ...);
int   vprintf(const char *format, va_list arg);
int   vsprintf(char *s, const char *format, va_list arg);
```

2.5.3 Debug API Functions

DbgPrintf — ***Print a Debug Message to the Debug Log***

Syntax

void DbgPrintf(int ErrLevel, char *Format, ?);

Parameters

ErrLevel	Severity level of the error
Format	Standard printf format string

Return Value

None.

Description

This function prints a debug message to the global debug log buffer. The log buffer is defined as follows:

```
#define LL_DEBUG_LOG_MAX        1024
extern char  DebugLog[LL_DEBUG_LOG_MAX];     // DebugLog Buffer
extern int   DebugLogSize;                   // Bytes of data currently in
DebugLog
```

The buffer behaves like one large NULL terminated string. The contents are cleared by setting *DebugLogSize* to 0.

The value of ErrLevel determines if the message is printed and additionally, if the message results in a system shutdown. Both of these thresholds (printing and shutdown) are set through the OS configuration. The definition of the severity levels are as follows:

```
#define DBG_INFO    1
#define DBG_WARN    2
#define DBG_ERROR   3
#define DBG_None    4
```

2.6 File I/O Support for Embedded Systems

2.6.1 Synopsis

The next section of this document discusses the support for stream IO that is built into the stack library. The support documented in that section is intended to augment the basic functions provided by the native operating system (in the case where the stack is ported to a new environment).

This section details functionality required by the Network Tools services interfacing with File IO. The functionality described here is more likely to have a local counterpart. The API described in this section must be ported to allow the network services that use it to operate.

The API described here was taken from the Unix standard. The names of the functions have been prefixed with the designation efs_ ,which stands for embedded file system. This was done so that the functions would not conflict with any existing file system. The EFS API is a very simple RAM based file system. A couple of new functions are included that allow the creation of RAM files by supplying pointers to static data buffers. For systems with existing file structures, most of the functions in this API become secondary to their standard IO counterparts.

> This API is unrelated to the stream API provided for Sockets. If the services that need this API are not required, then this module can be discarded from the OS abstraction. Currently, only the HTTP Server service uses this API.

2.6.2 Function Overview

The following functions are custom to this implementation, but can be ported:

efs_createfile()	Create (declare) RAM based file
efs_createfilecb()	Create (declare) RAM based file (with callback function)
efs_destroyfile()	Destroy RAM based file
efs_getfilesize()	Get the length of file data
efs_filecheck()	Check the file type and authorization
efs_filesend()	Send file contents directly to a socket
efs_loadfunction()	Load executable file and return entry-point function

As previously mentioned, most of the API closely matches its standard C counterpart:

efs_fclose()	Close file
efs_feof()	Check for end of file
efs_fopen()	Open file
efs_fread()	Read from file
efs_fseek()	Set file position
efs_ftell()	Get file position
efs_fwrite()	Write to file
efs_rewind()	Reset file position to start of file

2.6.3 EFS Custom API Functions

efs_createfile — *Create (declare) a RAM Based File*

Syntax

void efs_createfile(char *name, INT32 length, UINT8 *pData);

Parameters

name	Filename (maximum length of EFS_FILENAME_MAX)
length	Length of file data
pData	Pointer to file data

Return Value

None.

Description

This function creates an internal record of the RAM based file with the indicated filename, file length, and data pointer. The file data is not copied, so the buffer must be statically allocated. The filename is copied, so it does not need to be static.

A static buffer based system is more efficient for embedded systems because the data must already be present in RAM or ROM. However, the *efs_createfile()* function could easily be altered to use allocated buffers that are later freed when *efs_destroyfile()* is called. These create and destroy functions are only called by the sample application code, and thus the system programmer is free to alter the operation of these functions - so long as they create files that are compatible with the rest of this API.

efs_createfilecb — *Create (declare) a RAM Based File with Callback*

Syntax

void efs_createfilecb(char *name, INT32 length, UINT8 *pData, EFSFUN pcbFreeFun, UINT32 FreeArg);

Parameters

name	Filename (maximum length of EFS_FILENAME_MAX)
length	Length of file data
pData	Pointer to file data
pcbFreeFun	Pointer to file data
FreeArg	Pointer to file data

Return Value

None.

Description

This is identical to *efs_createfile()*, except that is takes two additional arguments, a pointer to a file free function, and a 32 bit argument. It is designed to be used in system where the memory used for the file is allocated, and not static.

The EFS file system tracks the numbers of references to a particular file. When the *efs_destroyfile()* function is called to destroy a file, the file is marked so that it can no longer be opened, but open handles to the file remain valid until closed by their respective application. The free function callback calls back to the file creator when the last file handle to the file has been closed, allowing the creator to safely reclaim any memory associated with the file. The argument FreeArg is used as a calling parameter to the callback.

efs_destroyfile — *Destroy (remove declaration from) a RAM Based File*

Syntax

void efs_destroyfile(char *name);

Parameters

name	Filename (maximum length of EFS_FILENAME_MAX)

Return Value

None.

Description

This function deletes the internal file record associating the filename with the static data pointer as originally passed to *efs_createfile()*.

A static buffer based system is more efficient for embedded systems because the data must already be present in RAM or ROM. However, the *efs_createfile()* function could easily be altered to use allocated buffers that are later freed when *efs_destroyfile()* is called. These create and destroy functions are only called by the sample application code, and thus the system programmer is free to alter the operation of these functions - so long as they create files that are compatible with the rest of this API.

efs_getfilesize — *Get the Length of a File*

Syntax

INT32 efs_getfilesize(EFS_FILE *stream);

Parameters

stream	Pointer to open stream (file)

Return Value

File size in bytes.

Description

This function returns the length in bytes of the indicated file. The file must already have been opened via a call to *efs_fopen()*.

efs_filecheck — *Check the file type and authorization*

Syntax

int efs_filecheck(char *name, char *user, char *password, int *prealm);

Parameters

name	Filename (NULL terminated string)
user	Username (NULL terminated string)
password	Password (NULL terminated string)
prealm	Pointer to receive realm Index (if authentication fails)

Return Value

An integer consisting of one or more of the following flags:

EFS_FC_NOTFOUND	File not found
EFS_FC_NOTALLOWED	File cannot be accessed
EFS_FC_EXECUTE	Filename represents a function call (CGI)
EFS_FC_AUTHFAILED	File authentication failed (failing realm Index supplied)

Description

This function is called by a file server (e.g., HTTP) on a particular filename (provided in *name*), to retrieve the file type, and authenticate user access. The user credentials are supplied in the user and password calling parameters.

The user and password arguments must always be valid pointers, but can be NULL strings.

When user authentication fails, the Index of the failing authentication realm (1 to 4) is written to the address supplied in *prealm*.

efs_filesend — *Send file contents directly to a socket*

Syntax

size_t efs_filesend(EFS_FILE *stream, size_t size, SOCKET s);

Parameters

stream	Pointer to open stream (file)
size	Number of bytes to transfer from the file
s	Socket onto which to send the file data

Return Value

Returns the number of bytes transferred, **NULL** on an error.

Description

This function is called by a file server (e.g., HTTP) on a particular file stream (provided in *stream*), to read data from the file and send it to socket *s*. Because EFS file systems are typically RAM based, this custom function can send the file to socket *s* more efficiently than an application that has to call *efs_read()* and then *send()*.

The number of bytes to transfer is given by *size*. Transfer begins and the current file pointer location, and the file pointer is advanced by this call.

efs_loadfunction — *Load Executable File and Return Entry-point*

Syntax

EFSFUN efs_loadfunction(char *name);

Parameters

name	Filename (maximum length of EFS_FILENAME_MAX)

Return Value

Pointer to executable function.

Description

This function loads an executable file and returns a pointer to the entry-point function. The type EFSFUN is declared as:

```
typedef void (*EFSFUN)();
```

The application is really free to treat this function in whatever manner is required. This executable file is created with a call to *efs_createfile()* where the *pData* parameter points to a function that is already loaded in memory. This allows the HTTP server to call services contained in CGI files.

A static buffer based system is more efficient for embedded systems because the data must already be present in RAM or ROM. However, the HTTP can be made to work with physical CGI files by porting this function to load CGI.

2.6.4 EFS Standard API Functions

efs_fclose — *Close File*

Syntax

int efs_fclose(EFS_FILE *stream);

Parameters

stream	Pointer to open stream (file)

Return Value

Returns EOF if any errors occurred, and zero otherwise.

Description

This function performs a logical close on an open file. It is functionally equivalent to *fclose()*.

efs_feof — *Test for End of File*

Syntax

int efs_feof(EFS_FILE *stream);

Parameters

stream	Pointer to open stream (file)

Return Value

Returns non-zero if EOF has been reached, and zero otherwise.

Description

This function tests to see is the file position has reached the end of the file. It is functionally equivalent to *feof()*.

efs_fopen — *Open File*

Syntax

EFS_FILE *efs_fopen(char *name, char *mode);

Parameters

name	Name of file to open
mode	Desired mode of open file

Return Value

Returns a stream pointer or NULL on error.

Description

This function performs a logical open on the named file and returns a stream or NULL if the attempt fails. It is functionally equivalent to *fopen()*.

The *mode* parameter determines the mode for which the file is opened. In the embedded file system version of this function, the list of supported modes is quite simple:

rb - open binary file for reading

The flags are still passed through to ensure compatibility with a full file system.

efs_fread — *Read from a File*

Syntax

size_t efs_fread(void *ptr, size_t size, size_t nobj, EFS_FILE *stream);

Parameters

ptr	Pointer to data buffer to receive data
size	Size in bytes of a read object
nobj	Number of objects to read
stream	Pointer to open stream (file)

Return Value

Returns the number of objects read.

Description

This function reads from the indicated *stream* in the array *ptr* at most *nobj* objects of a length specified by size. It returns the number of objects read; this may be less than the number of objects requested. It is functionally equivalent to *fread()*.

efs_feof() can be used to detect end of file.

efs_fseek — *Set File Position*

Syntax

INT32 efs_fseek(EFS_FILE *stream, INT32 offset, int origin);

Parameters

stream	Pointer to open stream (file)
offset	Offset of desired new position
origin	Base reference point for offset

Return Value

Returns non-zero on error.

Description

This function sets the file position of the indicated *stream* to that specified by *offset* from a base reference point specified by *origin*. It is functionally equivalent to *fseek()*.

The *origin* parameter can be set to one of the following:

- EFS_SEEK_SET - Position by *offset* from the beginning of the file
- EFS_SEEK_CUR - Position by *offset* from the current position
- EFS_SEEK_END - Position by *offset* from the end of the file

efs_ftell — *Get File Position*

Syntax

INT32 efs_ftell(EFS_FILE *stream);

Parameters

stream	Pointer to open stream (file)

Return Value

Returns file position or -1 on error.

Description

This function returns the current file position of the indicated stream. It is functionally equivalent to *ftell()*.

efs_fwrite — *Write to a File*

Syntax

size_t efs_fwrite(void *ptr, size_t size, size_t nobj, EFS_FILE *stream);

Parameters

ptr	Pointer to data buffer to receive data
size	Size in bytes of a read object
nobj	Number of objects to read
stream	Pointer to open stream (file)

Return Value

Returns the number of objects written (0).

Description

This function writes to the indicated stream from the array *ptr*, up to *nobj* objects of a length specified by *size*. It returns the number of objects written; this may be less than the number of objects requested on an error. It is functionally equivalent to *fwrite()*.

Nothing in the stack package requires write capability, thus this function always returns zero.

efs_rewind — *Reset File Position to Start of File*

Syntax

void efs_rewind(EFS_FILE *stream);

Parameters

stream	Pointer to open stream (file)

Return Value

None.

Description

This sets the position of the indicated *stream* to zero, and clears any current error. (Errors are not tracked in this implementation.)

Sockets and Stream IO API

This chapter describes the socket and file API functions.

Topic **Page**

3.1 File Descriptor Environment

In most embedded operating system environments, support for file descriptors varies greatly. In most cases, only the bare minimum functionality is provided, and trimmed down support functions are provided using the common reserved names (*read()*, *write()*, *close()*, etc.).

As this stack supports the standard sockets interface functions, and these functions require file descriptor support, the stack provides its own small file system. This section describes the basic mechanics of the file system.

3.1.1 Organization

The basic building block of the stack code internally is an object handle. Internally to the stack, both sockets and pipes are addressed by object handles. However, at the application level, sockets and pipes are treated as file descriptors. The file descriptor contains additional state information allowing tasks to be blocked and unblocked based on socket activity.

The stack API supports the use of file descriptors by adding a file descriptor layer of abstraction to the native operating environment. This layer implements the standard sockets and file IO functions. The stack works by associating a file descriptor session with each caller's thread (or in this terminology, task). In this system, each task has its own file descriptor session. The file descriptor session is used when the task needs to block pending network activity.

Note that although file descriptors can be used in classic functions like *select()*, in this implementation, they are still handles, not integers. For compatibility, network applications must use the NDK header files, and use INVALID_SOCKET for an error condition (not -1), and refrain from comparing sockets as <0 when checking for validity.

3.1.2 Initializing the File System Environment

To use the file system and socket functions provided by the stack, a task must first allocate a file descriptor table (called a file descriptor session). This is accomplished at the application layer by calling the file descriptor function *fdOpenSession()*.

When the task is finished using the file descriptor API, or when it is about to terminate, the function *fdCloseSession()* is called.

3.1.2.1 When to Initialize the File Descriptor Environment

For correct stack operation, a task thread must open a file descriptor session before calling any file descriptor related functions, and then close it when it is done.

The simplest way to handle the session is for the task to open a file session when it starts, and close the session when it completes. For example:

Socket Task:

```
void socket_task(int IPAddr, int TcpPort)
{
    SOCKET   s;

    // Open the file session
    fdOpenSession(TaskSelf());

    < socket application code >

    // Close the file session
    fdCloseSession(TaskSelf());
}
```

A second option is for the task that creates the socket task thread to open the file descriptor session for the child thread. Note that the parent task must guarantee that the child task's file session is open before the child task executes. This is done via task priority or semaphore, but can complicate task creation. Therefore, it is not the ideal approach.

A third, more common, option is to allow a child task to open its own file session, but allow the parent task to monitor its children and eventually destroy them. Here, the parent task must close the file session of the child task threads it destroys. The child task then blocks when finished instead of terminating its own thread. The following example illustrates this concept:

Child Socket Task:

```
void child_socket_task(int IPAddr, int TcpPort)
{
    SOCKET   s;
    // Open the file session

    fdOpenSession(TaskSelf());

    < socket application code >

    // We are done, but our parent thread will close
    // our file session and destroy this task, so here
    // we just block.
    TaskBlock(TaskSelf());
}
```

The parent task functions would look as follows:

Parent Task Functions:

```
void create_child_task()
{
    // Create System Tasks

    // Create a child task
    hChildTask = TaskCreate(&child_socket_task, ?);
}

void destroy_child_task()
{
    // First close the child's file session
    // (This will close all open files)
    fdSessionClose(hChildTask);

    // Then destroy the task
    TaskDestroy(hChildTask);
}
```

3.2 File Descriptor Programming Interface

3.2.1 Synopsis

The purpose of supporting a file system is to support the sockets API. Unfortunately, the sockets API is not a complete IO API, as it was originally designed to integrate into the Unix file system. Thus, several file descriptor functions that are important for application programming are not really socket calls at all. The stack library supports a handful of what are normally considered file functions, so that sockets applications can be programmed in a more traditional sense. So that these functions will not conflict with any other file functions in the system, their names have been altered slightly from the standard definitions.

3.2.2 Function Overview

The stream IO object can take two forms. In the vast majority of cases, it will be in the form of a local file descriptor. The following functions can operate on file descriptors:

fdOpenSession()	Open file descriptor support session
fdCloseSession()	Close file descriptor support session
fdClose()	Flush stream and close file descriptor (same as standard *close()*)
fdError()	Return last error value (same as standard *error*)
fdPoll()	Wait on a list of file descriptor events (same as standard *poll()*)
fdSelect()	Wait on one or more file events (same as standard *select()*)
fdSelectAbort()	Aborts calls to *fdSelect()* and *fdPoll()* with forced timeout condition
fdStatus()	Get the current status of a file descriptor (similar to *ioctl/FIONREAD*)
fdShare()	Add a reference count to a file descriptor

The *fdSelect()* function uses file descriptor sets to specify which file descriptors are being checked for activity and which have activity detected. There is a small set of MACRO functions for manipulating file descriptor sets. These include the following:

FD_SET()	Add a file descriptor to a file descriptor set
FD_CLR()	Remove a file descriptor from a file descriptor set
FD_ISSET()	Test to see if a file descriptor is included in a file descriptor set
FD_COPY()	Copy a file descriptor set
FD_ZERO()	Clear (initialize) a file descriptor set

3.2.3 File Descriptor API Functions

fdOpenSession — ***Open File Descriptor Session***

Syntax

int fdOpenSession(HANDLE hTask);

Parameters

hTask	Task Thread Handle

Return Value

1 on success or 0 on error. An error return indicates that a session is already open for the specified task, or that a memory allocation error has occurred.

Description

This function opens a file descriptor session on a task thread so that the task can begin using file descriptor and other stream IO functions.

A task thread normally calls *fdOpenSession()* when it is first created, and *fdCloseSession()* before it exits. Use of these functions was described in more detail in the previous section.

fdCloseSession *Close File Descriptor Session*

Syntax

void fdCloseSession(HANDLE hTask);

Parameters

hTask	Task Thread Handle

Return Value

None.

Description

This function closes a file descriptor session that was previously opened with *fdOpenSession()*. When called, any remaining open file descriptors are closed.

A task thread normally calls *fdOpenSession()* when it is first created, and *fdCloseSession()* before it exits. Use of these functions was described in more detail in the previous section.

fdClose *Close File Descriptor*

Syntax

int fdClose(HANDLE fd);

Parameters

fd	File Descriptor to close (compatible with type SOCKET)

Return Value

0 on success or -1 on error. When an error occurs, the error type can be obtained by calling *fdError()* (*error* is also equal to this function).

Description

This function closes the indicated file descriptor.

fdError *Get the Last File Error*

Syntax

int fdError();

Description

This function returns the last file error that occurred on the current task. In the SERRNO.H header file, *error* is equal to this function.

> **Note:** The error code returned via *fdError()* is stored in the file descriptor session associated with a task. If a task calls a file or socket function before it opens a file descriptor session, an error condition results. However, no error code can be stored for retrieval by *fdError()* because the file descriptor session does not exist to hold it.

fdPoll *Wait on a List of File Descriptor Events*

Syntax

int fdPoll(FDPOLLITEM items, uint itemcnt, INT32 timeout);

Parameters

items	Pointer to a list of descriptor events of type FDPOLLITEM
itemcnt	Number of entries in *items* list
timeout	Function timeout in milliseconds

Return Value

Returns the number of file descriptors in the items list for which the *eventsDetected* field is non-zero.

Returns SOCKET_ERROR if the caller has not opened a file descriptor session (with *fdOpenSession()*).

Returns zero (0) under any of the following conditions:

- No detected flags and time out has occurred
- No detected flags and a *fdSelectAbort()* was issued
- No detected flags and an internal resource allocation failed

Description

The *fdPoll()* function is a more efficient alternative to the *fdSelect()* function. It polls the supplied list of sockets, with a timeout specified in milliseconds (or POLLINFTIM for infinite timeout). It has the advantage over *fdSelect()* because the original list of file descriptors (or sockets) to be examined is not overwritten by the results, and thus can be used multiple times without reconstruction.

The list of file descriptors to check is provided in the *items* array. The array is of type FDPOLLITEM, which is defined as follows:

```
typedef struct _fdpollitem {
        HANDLE   fd;
        UINT16   eventsRequested;
        UINT16   eventsDetected;
    } FDPOLLITEM;
```

The FDPOLLITEM entry contains a file descriptor (or socket) to check, a set of flags for requested events that is initialized by the application, and a set of resulting flags for a detected event that is initialized by the *fdPoll()* function.

The entry *fd* is the file descriptor to check. If fd is set to INVALID_SOCKET, or the *eventsRequested* field is NULL, the item entry is ignored. However, the *eventsDetected* field is still reset to zero.

The same file descriptor should not appear twice in the list, instead the event flags should be combined on a single entry. (Duplicate descriptors will not cause an error, but will increase system load.)

Valid flags for *eventsRequested* are one or more of the following:

- POLLIN - Socket readable (or read error pending)
- POLLOUT - Socket writable (or send error pending)
- POLLPRI - Socket OOB readable (or error pending)
- POLLNVAL - Socket or request type invalid

Valid flags for *eventsDetected* are the same as above, where all detected conditions are indicated. (Note that POLLNVAL can be set whether or not it was requested in *eventsRequested*.)

fdSelect

Wait on one or multiple File Events

Syntax

int fdSelect(int maxfd, fd_set *readset, fd_set *writeset, fd_set *exceptset, struct timeval *timeout);

Parameters

maxfd	Ignored
readset	Set of file descriptors to check for reading
writeset	Set of file descriptors to check for writing
exceptset	Set of file descriptors to check for exceptional conditions (OOB data)
timeout	Pointer to timeval structure of time to wait (or NULL)

Return Value

Returns a positive count of ready descriptors (combined from all three possible sets), 0 on timeout, or -1 on error. When an error occurs, the error type can be obtained by calling *fdError()*.

Description

This function allows the task to instruct the stack to wait for any one of multiple events to occur and to wake up the process only when one of more of these events occurs or when a specified amount of time has passed.

The definition of the timeval structure is:

```
struct timeval {
    INT32 tv_sec;
    INT32 tv_usec;
};
```

Passing in a NULL pointer for timeout specifies an infinite wait period. Passing a valid pointer to a timeval structure with both *tv_sec* and *tv_usec* set to zero specifies that the function should not block.

Note: This function is less efficient than *fpPoll()*. In fact, the *fdSelect()* function calls *fdPoll()* after rearranging the descriptor sets into a *fdPoll()* descriptor list.

fdSelectAbort — *Terminate a Previous Call to fdSelect() or fdPoll()*

Syntax

void fdSelectAbort(HANDLE hTask);

Parameters

hTask	Handle to the task thread that is blocked in *fdSelect()* or *fdPoll()*

Return Value

None.

Description

This function aborts a call to *fdSelect()* or *fdPoll()* on the specified target thread by simulating a timeout condition (even when no timeout was originally specified). It can be used to wake a thread using a different method than socket or pipe activity. It is useful in callback functions where the handle to the target task thread is known, but where socket calls cannot be easily used.

The return value from the *fdSelect()* or *fdPoll()* function called on the target thread is still valid. In other words, if there is pending file descriptor activity, it will still be returned to the caller. However, if the target task thread is blocked in *fdSelect()* or *fdPoll()* at the time of the call, the most likely return value is zero for no activity.

If the target thread is not currently pending on a call to *fdSelect()* or *fdPoll()*, any subsequent call will be affected. Thus, the target thread is guaranteed to see the abort (although it may be accompanied by actual socket activity). So there is no race condition on calling *fdSelectAbort()* immediately prior to the target task thread calling *fdSelect()* or *fdPoll()*.

fdStatus — *Get the Current Status of a File Descriptor*

Syntax

int fdStatus(HANDLE fd, int request, int *results);

Parameters

fd	File descriptor (socket or pipe) to check
request	Status request type.
hTask	Pointer to where status results are written

Return Value

0 on success or -1 on error. When an error occurs, the error type can be obtained by calling *fdError()* (*errno* is also equal to this function).

Description

This function reads current status information about the file descriptor. The descriptor can be either a socket or a pipe object. The following describes the value written to *results* for the various *request* types and descriptor types:

- *request* = FDSTATUS_TYPE;

  The *results* pointer is written with the file descriptor type. It will be one of the following values:

  - FDSTATUS_TYPE_SOCKET - The file descriptor is a socket.
  - FDSTATUS_TYPE_PIPE - The file descriptor is a pipe.
- *request* = FDSTATUS_RECV;

  On listening sockets, the *results* pointer is written with:
  - -1 - There is an error pending on the socket.
  - 0 - There are no connections ready to be accepted.
  - 1 - There is at least one connection ready to be accepted.

  On data sockets, the *results* pointer is written with:
  - -1 - There is an error pending, or a call to *recv()* will result in an error.

    Note: On a TCP socket, this return value can also indicate that the peer connection has been closed and all available data has been read. In this case, a subsequent call to *recv()* will return NULL, not error.
  - <0 to n> - The number of bytes that can be read using *recv()* without blocking.
- *request* = FDSTATUS_SEND;

  On listening sockets, the *results* pointer is written with:
  - -1 - A listening socket can never be written.

  On TCP (non-ATOMIC) data sockets, the *results* pointer is written with:
  - -1 - There is an error pending, or a call to *send()* will result in an error.
  - <0 to n> - The number of bytes that can be written using *send()* without blocking.

  On UDP/RAW (ATOMIC) data sockets, the *results* pointer is written with:
  - -1 - There is an error pending, or a call to *send()* will result in an error.
  - <0 to n> - The maximum number of bytes that can be written using a single *send()* call.

fdShare — *Add a Reference Count to a File Descriptor*

Syntax

int fdShare(HANDLE fd);

Parameters

fd	File descriptor to share (compatible with type SOCKET)

Return Value

Returns zero on success or -1 on error.

Description

This is an optional function for applications that use descriptor sharing. It increments a reference count on the target descriptor, which is then decremented when the application calls *fdClose()*. It allows the descriptor to be shared among multiple tasks, each calling *fdClose()* when they are done, and the file descriptor is only closed by the final call. (Note that file descriptors are created with a reference call of 1, meaning that the first call to *fdClose()* will close the descriptor.)

3.2.4 File Descriptor Set (fd_set) Macros

FD_SET — *Add a File Descriptor to a File Descriptor Set*

Syntax

void FD_SET(HANDLE fd, fd_set *pFdSet);

Parameters

fd	File descriptor to add (compatible with type SOCKET)
pFdSet	Pointer to fd_set data type

Return Value

Should be treated as a void function. The true return value is dependent on the implementation of the macro.

Description

This function adds a file descriptor to a file descriptor set, typically before using the set in a call to *fdSelect()*. Note that after declaring a fd_set data type, it should be initialized using *FD_ZERO()* before attempting to set individual file descriptors.

FD_CLR — *Remove a File Descriptor From a File Descriptor Set*

Syntax

void FD_CLR(HANDLE fd, fd_set *pFdSet);

Parameters

fd	File descriptor to remove
pFdSet	Pointer to fd_set data type

Return Value

Should be treated as a void function. The true return value is dependent on the implementation of the macro.

Description

This function removes a file descriptor from a file descriptor set, typically after the file descriptor has been processed in a loop that continuously checks a file descriptor set.

FD_ISSET — *Test to See if a File Descriptor is Included in a File Descriptor Set*

Syntax

void FD_ISSET(HANDLE fd, fd_set *pFdSet);

Parameters

fd	File descriptor to check (compatible with type SOCKET)
pFdSet	Pointer to fd_set data type

Return Value

Returns an int value that should be treated as a TRUE/FALSE condition.

Description

This function returns TRUE if the supplied file descriptor is contained in the indicated file descriptor set. This function is typically called after a call to *fdSelect()* to determine on what file descriptors select has detected activity.

FD_COPY — *Copy a File Descriptor Set*

Syntax

void FD_COPY(fd_set *pFdSetSRC, fd_set *pFdSetDST);

Parameters

pFdSetSRC	Pointer to fd_set to copy
pFdSetDST	Pointer to fd_set to write copied data

Return Value None.

Description This function is called to make a copy of a file descriptor set. This is typically done if a set needs to be modified, but this original information needs to be maintained.

FD_ZERO *Clear (Initialize) a File Descriptor Set*

Syntax void FD_ZERO(fd_set *pFdSet);

Parameters

pFdSet	Pointer to fd_set to initialize

Return Value None.

Description This function is called to clear all bits in a file descriptor set. This should be the first call made on a newly declared fd_set variable.

3.3 Sockets Programming Interface

3.3.1 Synopsis

The socket function API supported by the stack library is consistent with the standard Berkeley sockets API. No parameter adjustments are required.

Two new types are defined for the socket function declarations:

```
typedef struct sockaddr      SA;
typedef struct sockaddr      *PSA;
```

3.3.2 Enhanced No-Copy Socket Operation

Any performance of any data stream operation suffers when data copies are performed. Although the stack software is designed to use a minimum number of data copies, memory efficiency and API compatibility sometimes require the use of data copy operations.

By default, neither UDP nor RAW sockets use send or receive buffers. However, the sockets API functions *recv()* and *recvfrom()* require a data buffer copy because of how the calling parameters to the functions are defined. In the stack library, two alternative functions (*recvnc()*and *recvncfrom()*) are provided to allow an application to get received data buffers directly without a copy operation. When the application is finished with these buffers, it returns them to the system via a call to *recvncfree()*.

By default, TCP uses both a send and receive buffer. The send buffer is used because the TCP protocol can require reshaping or retransmission of data due to window sizes, lost packets, etc. On receive, the standard TCP socket also has a receive buffer. This coalesces TCP data received from packet buffers. Coalescing data is important for protocols that transmit data in very small bursts (like a telnet session).

For TCP applications that get data in large bursts (and tend not to use flags like MSG_WAITALL on receive), the receive buffer can be eliminated by specifying an alternate TCP stream type of SOCK_STREAMNC (see *socket()*). Without the receive buffer, there is at least one less data copy because TCP will queue up the actual network packets containing receive data instead of copying it into a receive buffer.

Care needs to be taken when eliminating the TCP receive buffer. Here large amounts of packet buffers can be tied up for a small amount of data. Also, because packet buffers come from the HAL, there may be a limited supply available. If the MSG_WAITALL flag is used on a *recv()* or *recvfrom()* call, it is possible for all packet buffers to be consumed before the specified amount of payload data is received. This would cause a deadlock situation if no socket timeout is specified.

Although TCP sockets that use the SOCK_STREAMNC stream type are 100% compatible with the standard TCP socket type, they can also be used with the *recvnc()* and *recvncfrom()* functions that UDP and RAW sockets use to eliminate the final data copy from the stack to the sockets application. Using the no copy functions with SOCK_STREAMNC eliminates two data copies from the standard TCP socket. Note that when *recvnc()* and *recvncfrom()* are used with TCP, out of band data is not supported. If the SO_OOBINLINE socket option is set, the out of band data is retained, but the out of band data mark is discarded. If not using the inline socket option, the out of band data is discarded.

3.3.3 Function Overview

The standard socket access functions are as follows:

accept()	Accept a connection on a socket
bind()	Bind a name to a socket
connect()	Initiate a connection on a socket
getpeername()	Return name (address) of connected peer
getsockname()	Return the local name (address) of the socket
getsockopt()	Get the value of a socket option
listen()	Listen for connection requests on a socket
recv()	Receive data from a socket
recvfrom()	Receive data from a socket with the senders name (address)
send()	Send data to a connected socket
sendto()	Send data to a specified destination on an unconnected socket
setsockopt()	Set the value of a socket option
shutdown()	Close one half of a socket connection
socket()	Create a socket
socketpair()	Create socket pair (redundant; see Section 3.4, *Full Duplex Pipes Programming Interface*)

The enhanced socket functions are as follows:

recvnc()	Receive no-copy data from a socket
recvncfree()	Free buffer obtained from *recvnc()* or *recvncfrom()*
recvncfrom()	Receive no-copy data from a socket with the senders name (address)

3.3.4 Sockets API Functions

accept ***Accept a Connection on a Socket***

Syntax

SOCKET accept(SOCKET s, PSA pName, int *plen);

Parameters

s	Socket
pName	Name (address) of connected peer
plen	Pointer to size of pName

Return Value

If it succeeds, the function returns a non-negative integer that is a descriptor for the accepted socket. Otherwise, a value of INVALID_SOCKET is returned and the function *fdError()* can be called to determine the error:

EBADF	The file descriptor (socket) is invalid.
ECONNABORTED	Listening socket has been shut down for read operations.
EMFILE	The file descriptor table is full.
ENOMEM	Memory allocation error.
ENOTSOCK	The descriptor does not reference a socket.
EINVAL	*listen()* has not been called on the socket or name arguments are invalid.
EWOULDBLOCK	Socket is marked non-blocking and no connections are ready

Description

The argument *s* is a socket that has been created with the *socket()* function, bound to an address with *bind()*, and is listening for connections after a *listen()*. The *accept()* function extracts the first connection request on the queue of pending connections, creates a new socket with the same properties of socket *s* and allocates a new file descriptor for the socket. If no pending connections are present on the queue, and the socket is not marked as non-blocking, accept blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, accept returns an error as described above.

The accepted socket may not be used to accept more connections. The original socket *s* remains open.

The argument *pName* is a result parameter that is filled in with the address of the connecting entity as known to the communications layer. The domain in which the communication is occurring determines the exact format of the *pName* parameter. The *plen* is a value-result parameter; it should initially contain at least sizeof(struct sockaddr), the amount of space pointed to by *pName*; on return it will contain the actual length (in bytes) of the address returned.

This call is used with connection-based socket types, currently with SOCK_STREAM.

It is possible to select (*fdSelect()*) a socket for the purposes of doing an accept by selecting it for read.

bind

Bind a Name (Address) to a Socket

Syntax

int bind(SOCKET s, PSA pName, int len);

Parameters

s	Socket
pName	Name (address) of desired local address
len	Size of pName

Return Value

If it succeeds, the function returns 0. Otherwise, a value of -1 is returned and the function *fdError()* can be called to determine the error:

EBADF	The file descriptor (socket) is invalid.
ENOTSOCK	The descriptor does not reference a socket.
EINVAL	Name arguments are invalid.
EADDRNOTAVAIL	The specified address is not available from the local machine.
EADDRINUSE	The specified address is already in use.

Description

The *bind()* function assigns a name to an unnamed socket. When a socket is created with *socket()* it exists in a name space (address family) but has no name assigned. The *bind()* function requests that name be assigned to the socket.

The argument *s* is a socket that has been created with the *socket()* function. The argument *pName* is a structure of type sockaddr that contains the desired local address. The *len* parameter contains the size of *pName,* which is sizeof(struct sockaddr).

connect

Initiate a Connection on a Socket

Syntax

int connect(SOCKET s, PSA pName, int len);

Parameters

s	Socket
pName	Name (address) of desired peer
len	Size of pName

Return Value

If it succeeds, the function returns 0. Otherwise, a value of -1 is returned and the function *fdError()* can be called to determine the error:

EADDRINUSE	The specified address is already in use.
EADDRNOTAVAIL	The specified address is not available from the local machine.
EALREADY	A connection request is already pending on this socket.
EBADF	The file descriptor (socket) is invalid.
ECONNREFUSED	The attempt to connect was forcefully rejected.
EHOSTUNREACH	The host is not reachable.
EINPROGRESS	The request was accepted and is pending (non-blocking sockets).
EINVAL	Name arguments are invalid.
EISCONN	The socket is already connected.
ENOTSOCK	The file descriptor does not reference a socket.
ENOTSUPP	Socket is in the listening state and cannot be connected.
ETIMEDOUT	Connection establishment timed out without establishing a connection.

Description

The *connect()* function establishes a logical (and potentially physical) connection from the socket specified by *s* to the foreign name (address) specified by *pName*.

If sock is of type SOCK_DGRAM, this call specifies the peer address with which the socket is to be associated; this address is that to which datagrams are to be sent, and the only address from which datagrams are to be received. If the socket is of type SOCK_STREAM, the function attempts to make a connection to another socket.

The argument *s* is a socket that has been created with the *socket()* function. The argument *pName* is a structure of type sockaddr that contains the desired foreign address. The *len* parameter contains the size of *pName*, which is sizeof(struct sockaddr).

Stream sockets may connect only once; while datagram sockets may re-connect multiple times to change their association. The connection may be dissolved by attempting to connect to an illegal address (for example, NULL IP address and Port). Datagram sockets that require multiple connections may consider using the *recvfrom()* and *sendto()* functions instead of *connect()*.

It is possible to select (*fdSelect()*) a socket for the purposes of doing a connect by

selecting it for writing.

getpeername — *Get Name (Address) of Connected Peer*

Syntax

int getpeername(SOCKET s, PSA pName, int *plen);

Parameters

s	Socket
pName	Name (address) of connected peer
plen	Pointer to size of pName

Return Value

If it succeeds, the function returns 0. Otherwise, a value of -1 is returned and the function *fdError()* can be called to determine the error:

EBADF	The file descriptor (socket) is invalid.
ENOTSOCK	The file descriptor does not reference a socket.
EINVAL	Name arguments are invalid.
ENOTCONN	The socket is not connected.

Description

The *getpeername()* function returns the name (address) of the connected peer.

The argument *pName* is a result parameter that is filled in with the address of the connecting entity as known to the communications layer. The domain in which the communication is occurring determines the exact format of the *pName* parameter. The *plen* is a value-result parameter; it should initially contain at least sizeof(struct sockaddr), the amount of space pointed to by *pName*; on return it will contain the actual length (in bytes) of the address returned.

getsockname — *Get the Local Name (Address) of the Socket*

Syntax

int getsockname(SOCKET s, PSA pName, int *plen);

Parameters

s	Socket
pName	Name (address) of connected peer
plen	Pointer to size of pName

Return Value

If it succeeds, the function returns 0. Otherwise, a value of -1 is returned and the function *fdError()* can be called to determine the error:

EBADF	The file descriptor (socket) is invalid.
ENOTSOCK	The file descriptor does not reference a socket.
EINVAL	Name arguments are invalid.

Description

The *getsockname()* function returns the local name (address) of the socket.

The argument *pName* is a result parameter that is filled in with the address of the connecting entity as known to the communications layer. The domain in which the communication is occurring determines the exact format of the *pName* parameter. The *plen* is a value-result parameter; it should initially contain at least sizeof(struct sockaddr), the amount of space pointed to by *pName*; on return it will contain the actual length (in bytes) of the address returned.

getsockopt — *Get the Value of a Socket Option Parameter*

Syntax

int getsockopt(SOCKET s, int level, int op, void *pbuf, int *pbufsize);

Parameters

s	Socket
level	Option level (SOL_SOCKET, IPPROTO_IP, IPPROTO_TCP)
op	Socket option to get
pbuf	Pointer to memory buffer
pbufsize	Pointer to size of memory buffer

Return Value

If it succeeds, the function returns 0. Otherwise, a value of -1 is returned and the function *fdError()* can be called to determine the error:

EBADF	The file descriptor (socket) is invalid.
ENOTSOCK	The file descriptor does not reference a socket.
EINVAL	Buffer arguments are invalid.

Description

The *getsockopt()* function returns the options associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost socket level.

When manipulating socket options, the level at which the option resides and the name of the option must be specified. To manipulate options at the socket level, level is specified as SOL_SOCKET. To manipulate options at any other level, the protocol number of the appropriate protocol controlling the option is supplied. In this implementation, only SOL_SOCKET, IPPROTO_IP, and IPROTO_TCP are supported.

The parameters *pbuf* and *pbufsize* identify a buffer in which the value for the requested option(s) are to be returned. *pbufsize* is a value-result parameter, initially containing the size of the buffer pointed to by *pbuf*, and modified on return to indicate the actual size of the value returned.

Most socket-level options utilize an int parameter for *pbuf*. SO_LINGER uses a struct linger parameter, defined in INC\SOCKET.H, which specifies the desired state of the option and the linger interval (see below). SO_SNDTIMEO and SO_RCVTIMEO use a struct timeval parameter.

The following options are recognized at the socket level:

SO_REUSEADDR	Specifies that the rules used in validating addresses supplied in a bind call should allow reuse of local addresses.
SO_REUSEPORT	Allows completely duplicate bindings by multiple processes if they all set SO_REUSEPORT before binding the port. This option permits multiple instances of a program to each receive UDP/IP multicast or broadcast datagrams destined for the bound port.
SO_KEEPALIVE	Enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken and processes using the socket are notified when attempting to send data.
SO_DONTROUTE	Indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

SO_LINGER	Controls the action taken when unsent messages are queued on socket and a close is performed. If the socket promises reliable delivery of data and SO_LINGER is set, the system will block the process on the close attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, termed the linger interval, is specified in seconds in the setsockopt call when SO_LINGER is requested). If SO_LINGER is disabled and a close is issued, the system will process the close in a manner that allows the process to continue as quickly as possible.
SO_BROADCAST	Requests permission to send broadcast datagrams on the socket. Broadcast was a privileged operation in earlier versions of the system.
SO_OOBINLINE	With protocols that support out-of-band data, this option requests that out-of-band data be placed in the normal data input queue as received; it will then be accessible with recv or read calls without the MSG_OOB flag. Some protocols always behave as if this option is set.
SO_SNDBUF	Buffer size for output.
SO_RCVBUF	Buffer size for input.
SO_SNDLOWAT	Is an option to set the minimum count for output operations. Most output operations process all of the data supplied by the call, delivering data to the protocol for transmission and blocking as necessary for flow control. Non-blocking output operations will process as much data as permitted subject to flow control without blocking, but will process no data if flow control does not allow the smaller of the low water mark value or the entire request to be processed. A select operation testing the ability to write to a socket will return true only if the low water mark amount could be processed. The default value for SO_SNDLOWAT is set to a convenient size for network efficiency, often 1024.
SO_RCVLOWAT	Is an option to set the minimum count for input operations. In general, receive calls will block until any (non-zero) amount of data is received, then return with the smaller of the amount specified by SO_RCVLOWAT or the amount requested. The default value for SO_RCVLOWAT is 1. Receive calls may still return less than the amount specified by SO_RCVLOWAT or the amount requested if an error occurs, or the type of data next in the receive queue is different from that which was returned.
SO_SNDTIMEO	Is an option to set a timeout value for output operations. It accepts a struct timeval parameter with the number of seconds and microseconds used to limit waits for output operations to complete. If a send operation has blocked for this much time, it returns with a partial count or with the error EWOULDBLOCK if no data were sent. In the current implementation, this timer is restarted each time additional data are delivered to the protocol, implying that the limit applies to output portions ranging in size from the low water mark to the high water mark for output.

SO_RCVTIMEO	Is an option to set a timeout value for input operations. It accepts a struct timeval parameter with the number of seconds and microseconds used to limit waits for input operations to complete. This timer is restarted each time additional data are received by the protocol, and thus, the limit is in effect an inactivity timer. If a receive operation has been blocked for this much time without receiving additional data, it returns with a short count or with the error EWOULDBLOCK if no data were received.
SO_TYPE	SO_TYPE returns the type of the socket, such as SOCK_STREAM.
SO_ERROR	Returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

Options that are not Berkeley standard:

SO_IFDEVICE	Specifies a uint index (1 to *n*) of the designated interface for sending and receiving IP broadcast packets. When set, this interface is selected on a IP broadcast send operation if the socket's local (bound) IP address is NULL (INADDR_ANY). Also, when set, the socket will only accept incoming broadcast packets if they have been received on this interface.
SO_BLOCKING	Specifies a int flag (1 or 0) indicating if the socket is in blocking or non-blocking mode. Sockets default to blocking mode when created, but can be set to non-blocking by using *setsockopt()*. This option provides the same functionality as calling the Unix function *Fcntl()* with the O_NONBLOCK flag.

The following options are recognized at the IPPROTO_IP level:

IP_OPTIONS	Specifies the IP options to be included in any outgoing IP packet sent via this socket (maximum length is 20 bytes).
IP_HDRINCL	Indicates to IP that the socket application is supplying the IP header as well as the rest of the packet payload. This is for use with RAW sockets only.
IP_TOS	Specifies the TOS value to place in the IP header.
IP_TTL	Specifies the TTLvalue to place in the IP header.

The following options are recognized at the IPPROTO_TCP level:

TCP_MAXSEG	Set the maximum TCP segment size.
TCP_NODELAY	Disables TCP send delay/coalesce algorithm.
TCP_NOPUSH	Do not send data just to finish a data block (attempt to coalesce).
TCP_NOOPT	Do not use TCP options.

listen — *Listen for Connection Requests on Socket*

Syntax

int listen(SOCKET s, int maxcon);

Parameters

s	Socket
maxcon	Maximum number of connects to queue

Return Value

If it succeeds, the function returns 0. Otherwise, a value of -1 is returned and the function *fdError()* can be called to determine the error:

EBADF	The file descriptor (socket) is invalid.
ENOTSOCK	The file descriptor does not reference a socket.
EOPNOTSUPP	The socket is not of a type that supports the operation listen.
EISCONN	The socket is already connected

Description

The *listen()* function listens for connection requests on a socket.

To accept connections, a socket is first created with *socket()*. The *listen()* function is called to specify a willingness to accept incoming connections and a queue limit for incoming connections. New connections are accepted by calling the *accept()* function. The *listen()* function applies only to sockets of type SOCK_STREAM.

The *maxcon* parameter defines the maximum length to which the queue of pending connections may grow. If a connection request arrives with the queue full, the client receives an error with an indication of ECONNREFUSED.

recv — *Receive Data from a Socket*

Syntax

int recv(SOCKET s, void *pbuf, int size, int flags);

Parameters

s	Socket
pbuf	Data buffer to place received data
size	Size of desired data
flags	Option flags

Return Value

If it succeeds, the function returns the number of bytes received. Returns 0 on connection oriented sockets where the connection has been closed by the peer (or socket shutdown for read). Otherwise, a value of -1 is returned and the function *fdError()* can be called to determine the error:

EBADF	The file descriptor (socket) is invalid.
EINVAL	Attempt to read (or calling arguments) invalid for this socket.
ENOTCONN	The socket is connection oriented and not connected
ENOTSOCK	The file descriptor does not reference a socket.
ETIMEDOUT	The socket connection was dropped due to protocol layer timeout.
EWOULDBLOCK	The socket is specified as non-blocking, or the timeout has expired.

Description

The *recv()* function attempts to receive data from a socket. It is normally used on a connected socket (see *connect()*). The data is placed into the buffer specified by *pbuf*, up to a maximum length specified by size. The options in flags can be used to change the default behavior of the operation.

The functions returns the length of the message on successful completion.

For a datagram type socket, the receive operation always copies one packet's worth of data. If the buffer is too short to hold the entire packet, the data is truncated and lost.

If no messages are available at the socket, it waits for a message to arrive, unless the socket is non-blocking. The function normally returns any data available, up to the requested amount, rather than waiting for receipt of the full amount requested; this behavior is affected by the options specified in flags as well as the socket-level options SO_RCVLOWAT and SO_RCVTIMEO described in *getsockopt()* .

The select call (*fdSelect()*) may be used to determine when more data arrives.

The *flags* argument to a *recv()* call is formed by combining one or more of the following flags:

MSG_DONTWAIT	Requests that the operation not block when no data is available.
MSG_OOB	Requests receipt of out-of-band data that would not be received in the normal data stream. Some protocols place expedited data at the head of the normal data queue, and thus, this flag cannot be used with such protocols.
MSG_PEEK	Causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, a subsequent receive call will return the same data.
MSG_WAITALL	Requests that the operation block until the full request is satisfied. However, the call may still return less data than requested if an error or disconnect occurs, or the next data to be received is of a different type than that returned.

recvfrom — *Receive Data from a Socket with the Sender's Name (Address)*

Syntax

int recvfrom(SOCKET s, void *pbuf, int size, int flags, PSA pName, int *plen);

Parameters

s	Socket
pbuf	Data buffer to place received data
size	Size of desired data
flags	Option flags
pName	Pointer to place name (address) of sender
plen	Pointer to size of pName

Return Value

If it succeeds, the function returns the number of bytes received. Returns 0 on connection oriented sockets where the connection has been closed by the peer (or socket shutdown for read). Otherwise, a value of -1 is returned and the function *fdError()* can be called to determine the error:

EBADF	The file descriptor (socket) is invalid.
EINVAL	Attempt to read (or calling arguments) invalid for this socket.
ENOTCONN	The socket is connection oriented and not connected.
ENOTSOCK	The file descriptor does not reference a socket.
ETIMEDOUT	The socket connection was dropped due to protocol layer timeout.
EWOULDBLOCK	The socket is specified as non-blocking, or the timeout has expired.

Description

The *recvfrom()* function attempts to receive data from a socket. It is normally called with unconnected, non-connection oriented sockets. The data is placed into the buffer specified by *pbuf*, up to a maximum length specified by size. The options in flags can be used to change the default behavior of the operation. The name (address) of the sender is written to *pName*.

The argument *pName* is a result parameter that is filled in with the address of the sending entity as known to the communications layer. The domain in which the communication is occurring determines the exact format of the *pName* parameter. The *plen* is a value-result parameter; it should initially contain at least sizeof(struct sockaddr),

the amount of space pointed to by pName; on return it will contain the actual length (in bytes) of the address returned.

The functions returns the length of the message on successful completion.

For a datagram type socket, the receive operation always copies one packet's worth of data. If the buffer is too short to hold the entire packet, the data is truncated and lost.

If no messages are available at the socket, it waits for a message to arrive, unless the socket is non-blocking. The function normally returns any data available, up to the requested amount, rather than waiting for receipt of the full amount requested; this behavior is affected by the options specified in flags as well as the socket-level options SO_RCVLOWAT and SO_RCVTIMEO described in *getsockopt()* .

The select call (*fdSelect()*) may be used to determine when more data arrives.

The flags argument to a *recv()* call is formed by combining one or more of the following flags:

MSG_DONTWAIT	Requests that the operation not block when no data is available.
MSG_OOB	Requests receipt of out-of-band data that would not be received in the normal data stream. Some protocols place expedited data at the head of the normal data queue, and thus, this flag cannot be used with such protocols.
MSG_PEEK	Causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, a subsequent receive call will return the same data.
MSG_WAITALL	Requests that the operation block until the full request is satisfied. However, the call may still return less data than requested if an error or disconnect occurs, or the next data to be received is of a different type than that returned.

recvnc *Receive Data from a Socket without Buffer Copy*

Syntax

int recvnc(SOCKET s, void **ppbuf, int flags, HANDLE *phBuffer);

Parameters

s	Socket
ppbuf	Pointer to receive data buffer pointer
flags	Option flags
phBuffer	Pointer to receive buffer handle

Return Value

If it succeeds, the function returns the number of bytes received. Returns 0 on connection oriented sockets where the connection has been closed by the peer (or socket shutdown for read). Otherwise, a value of -1 is returned and the function *fdError()* can be called to determine the error:

EBADF	The file descriptor (socket) is invalid.
EINVAL	Attempt to read (or calling arguments) invalid for this socket.
ENOTSOCK	The file descriptor does not reference a socket.
ENOTCONN	The socket is connection oriented and not connected.
ETIMEDOUT	The socket connection was dropped due to protocol layer timeout.
EWOULDBLOCK	The socket is specified as non-blocking, or the timeout has expired.

Description

The *recvnc()* function attempts to receive a data buffer from a socket. It is normally used on a connected socket (see *connect()*). A pointer to the data buffer is returned in *ppbuf*. A system handle used to free the buffer is returned in *phBuffer*. Both of these pointers must be valid. The options in flags can be used to change the default behavior of the operation.

The functions returns the length of the message on successful completion.

The receive operation always returns one packet buffer. The caller has no control over the size of the data returned in this buffer.

If no messages are available at the socket, this call waits for a message to arrive, unless the socket is non-blocking. The function returns the data buffer available.

When the caller no longer needs the data buffer, it is returned to the system by calling *recvncfree()*. Repeated failure to free buffers will eventually cause the stack to stop receiving data.

This function cannot be used with sockets of type SOCK_STREAM. When used with sockets of type SOCK_STREAMNC, out of band data marks are cleared.

The select call (*fdSelect()*) may be used to determine when more data arrives.

The flags argument to a *recv()* call can be one of the following flags:

MSG_DONTWAIT	Requests that the operation not block when no data is available.
MSG_WAITALL	Requests that the operation block until data is available. Because blocking is the default behavior of a standard socket, this flag only alters the behavior of a non blocking socket for this call.

recvncfree — *Return a Data Buffer Obtained from a No-Copy Receive Operation*

Syntax

void recvncfree(HANDLE hBuffer);

Parameters

hBuffer	Handle to receive buffer to free

Return Value

None.

Description

The *recvncfree()* function frees a data buffer obtained from calling either *recvnc()* or *recvncfrom()*. The calling parameter *hBuffer* is the handle of the buffer to free (not the pointer to the buffer).

recvncfrom — *Receive Data and the Sender's Name From a Socket Without Buffer Copy*

Syntax

int recvncfrom(SOCKET s, void **ppbuf, int flags, PSA pName, int *plen, HANDLE *phBuffer);

Parameters

s	Socket
ppbuf	Pointer to receive data buffer pointer
flags	Option flags
pName	Pointer to place name (address) of sender
plen	Pointer to size of pName
phBuffer	Pointer to receive buffer handle

Return Value

If it succeeds, the function returns the number of bytes received. Returns 0 on connection oriented sockets where the connection has been closed by the peer (or socket shutdown for read). Otherwise, a value of -1 is returned and the function *fdError()* can be called to determine the error:

EBADF	The file descriptor (socket) is invalid.
EINVAL	Attempt to read (or calling arguments) invalid for this socket.
ENOTSOCK	The file descriptor does not reference a socket.
ENOTCONN	The socket is connection oriented and not connected.
ETIMEDOUT	The socket connection was dropped due to protocol layer timeout.
EWOULDBLOCK	The socket is specified as non-blocking, or the timeout has expired.

Description

The *recvncfrom()* function attempts to receive a data buffer from a socket. It is normally called with unconnected, non-connection oriented sockets. A pointer to the data buffer is returned in *ppbuf*. A system handle used to free the buffer is returned in *phBuffer*. Both of these pointers must be valid. The options in flags can be used to change the default behavior of the operation. The name (address) of the sender is written to *pName*.

The argument *pName* is a result parameter that is filled in with the address of the sending entity as known to the communications layer. The domain in which the communication is occurring determines the exact format of the *pName* parameter. The *plen* is a value-result parameter; it should initially contain at least sizeof(struct sockaddr), the amount of space pointed to by *pName*; on return it will contain the actual length (in bytes) of the address returned.

The functions returns the length of the message on successful completion.

The receive operation always returns one packet buffer. The caller has no control over the size of the data returned in this buffer.

If no messages are available at the socket, this call waits for a message to arrive, unless the socket is non-blocking. The function returns the data buffer available.

When the caller no longer needs the data buffer, it is returned to the system by calling *recvncfree()*. Repeated failure to free buffers will eventually cause the stack to stop receiving data.

This function cannot be used with sockets of type SOCK_STREAM. When used with sockets of type SOCK_STREAMNC, out of band data marks are cleared.

The select call (*fdSelect()*) may be used to determine when more data arrives.

The *flags* argument to a *recv()* call can be one of the following flags:

MSG_DONTWAIT	Requests that the operation not block when no data is available.
MSG_WAITALL	Requests that the operation block until data is available. Because blocking is the default behavior of a standard socket, this flag only alters the behavior of a non blocking socket for this call.

send

Transmit Data to a Socket

Syntax

int send(SOCKET s, void *pbuf, int size, int flags);

Parameters

s	Socket
pbuf	Data buffer holding data to transmit
size	Size of data
flags	Option flags

Return Value

If it succeeds, the function returns the number of bytes sent. Otherwise, a value of -1 is returned and the function fdError() can be called to determine the error:

EBADF	The file descriptor (socket) is invalid.
EHOSTUNREACH	The remote host was unreachable.
EMSGSIZE	The specified size exceeds the limit of the underlying protocol.
ENOBUFS	Memory allocation failure while attempting to send data.
ENOTSOCK	The file descriptor does not reference a socket.
ENOTCONN	The socket is connection oriented and not connected.
ESHUTDOWN	The socket has been shut down for writes.
ETIMEDOUT	The socket connection was dropped due to protocol layer timeout.
EWOULDBLOCK	The socket is specified as non-blocking, or the timeout has expired.

Description

The *send()* function attempts to send data on a socket. It is used on connected sockets only (see *connect()*). The data to send is contained in the buffer specified by *pbuf*, with a length specified by size. The options in flags can be used to change the default behavior of the operation.

The functions returns the length of the data transmitted on successful completion.

For a datagram type socket, the send operation always copies one packet's worth of data. If the buffer size is too large to be transmitted in a single packet, an error code of EMSGSIZE is returned.

If there is not transmit buffer space available on a stream type socket, the function waits for space to become available, unless the socket is non-blocking. The function normally transmits all the specified data.

The select call (*fdSelect()*) may be used to determine when the socket is able to write.

The flags argument to a *send()* call is formed by combining one or more of the following flags:

MSG_OOB	sends out-of-band data on sockets that support this notion (e.g. SOCK_STREAM); the underlying protocol must also support out-of-band data.
MSG_EOR	indicates a record mark for protocols that support the concept.
MSG_EOF	Requests that the sender side of a socket be shut down, and that an appropriate indication be sent at the end of the specified data; this flag is only implemented for SOCK_STREAM sockets in the PF_INET protocol family, and implements Transaction TCP.
MSG_DONTROUTE	Specifies that the packet should not be routed, but sent only using the ARP table entries.

sendto — *Transmit Data on a Socket to Designated Destination*

Syntax

int sendto(SOCKET s, void *pbuf, int size, int flags, PSA pName, int len);

Parameters

s	Socket
pbuf	Data buffer holding data to transmit
size	Size of data
flags	Option flags
pName	Pointer to name (address) of destination
len	Size of data pointed to by pName

Return Value

If it succeeds, the function returns the number of bytes sent. Otherwise, a value of -1 is returned and the function *fdError()* can be called to determine the error:

EBADF	The file descriptor (socket) is invalid.
EHOSTUNREACH	The remote host was unreachable.
EMSGSIZE	The specified size exceeds the limit of the underlying protocol.
ENOBUFS	Memory allocation failure while attempting to send data.
ENOTSOCK	The file descriptor does not reference a socket.
ENOTCONN	The socket is connection oriented and not connected.
ESHUTDOWN	The socket has been shut down for writes.
ETIMEDOUT	The socket connection was dropped due to protocol layer timeout.
EWOULDBLOCK	The socket is specified as non-blocking, or the timeout has expired.

Description

The *sendto()* function attempts to send data on a socket to a specified destination. It is used on unconnected, non-connection oriented sockets only (see *connect()*). The data to send is contained in the buffer specified by pbuf, with a length specified by size. The options in flags can be used to change the default behavior of the operation.

The argument *pName* is a pointer to the address of the destination entity as known to the communications layer. The domain in which the communication is occurring determines the exact format of the *pName* parameter. The *len* parameter should contain the size of name, which is sizeof(struct sockaddr).

The functions returns the length of the data transmitted on successful completion.

For a datagram type socket, the send operation always copies one packet's worth of data. If the buffer size is too large to be transmitted in a single packet, an error code of EMSGSIZE is returned.

The select call (*fdSelect()*) may be used to determine when the socket is able to write.

The flags argument to a *send()* call is formed by combining one or more of the following flags:

MSG_OOB	sends out-of-band data on sockets that support this notion (e.g., SOCK_STREAM); the underlying protocol must also support out-of-band data.
MSG_EOR	indicates a record mark for protocols that support the concept.

MSG_EOF	Requests that the sender side of a socket be shut down, and that an appropriate indication be sent at the end of the specified data; this flag is only implemented for SOCK_S TREAM sockets in the PF_INET protocol family, and implements Transaction TCP.
MSG_DONTROUTE	Specifies that the packet should not be routed, but sent only using the ARP table entries.

setsockopt — *Set the Value of a Socket Option Parameter*

Syntax

int setsockopt(SOCKET s, int level, int op, void *pbuf, int bufsize);

Parameters

s	Socket
level	Option level (SOL_SOCKET, IPPROTO_IP, IPPROTO_TCP)
op	Socket option to get
pbuf	Pointer to memory buffer
bufsize	Size of memory buffer pointed to by pbuf

Return Value

If it succeeds, the function returns 0. Otherwise, a value of -1 is returned and the function *fdError()* can be called to determine the error:

EBADF	The file descriptor (socket) is invalid.
ENOTSOCK	The file descriptor does not reference a socket.
EINVAL	Buffer arguments are invalid.

Description

The *setsockopt()* function sets option values associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost socket level.

When manipulating socket options, the level at which the option resides and the name of the option must be specified. To manipulate options at the socket level, level is specified as SOL_SOCKET. To manipulate options at any other level, the protocol number of the appropriate protocol controlling the option is supplied. In this implementation, only SOL_SOCKET, IPPROTO_IP, and IPROTO_TCP are supported.

The parameters *pbuf* and *bufsize* identify a buffer that holds the value for the specified option.

Most socket-level options utilize an int parameter for *pbuf*. SO_LINGER uses a struct linger parameter, defined in INC\SOCKET.H, which specifies the desired state of the option and the linger interval. SO_SNDTIMEO and SO_RCVTIMEO use a struct timeval parameter.

The socket options supported for *setsockopt()* are the same as defined for *getsockopt()*, with the exception of SO_TYPE and SO_ERROR, which cannot be set.

Please see the description of getsockopt() for a list of socket options.

Note: The SO_SNDBUFand SO_RCVBUFoptions can only be set if there is no transmit or receive data pending at the socket. In general, the buffer sizes should only be configured before the socket is bound or connected. Buffer sizes set on listen sockets will propagate to spawned accept sockets.

shutdown — *Close One Half of a Connected Socket*

Syntax

int shutdown(SOCKET s, int how);

Parameters

s	Socket
how	Manner of shut down

Return Value

If it succeeds, the function returns 0. Otherwise, a value of -1 is returned and the function *fdError()* can be called to determine the error:

EBADF	The file descriptor (socket) is invalid.
ENOTSOCK	The file descriptor does not reference a socket.
ENOTCONN	The specified socket is not connected.

Description

The *shutdown()* function causes all or part of a full-duplex connection on the socket associated with a socket to be shut down. If *how* is SHUT_RD (0), further receives will be disallowed. If *how* is SHUT_WR (1), further sends will be disallowed. If *how* is SHUT_RDWR (2), further sends and receives will be disallowed.

socket — *Create a Socket*

Syntax

SOCKET socket(int domain, int type, int protocol);

Parameters

domain	Socket domain (PF_INET)
type	Socket type (SOCK_DGRAM, SOCK_STREAM, SOCK_RAW)
protocol	Socket protocol (Normally IPPROTO_TCP or IPPROTO_UDP, but can be anything when type is set to SOCK_RAW)

Return Value

If it succeeds, the function returns a file descriptor representing the socket. Otherwise, a value of INVALID_SOCKET is returned and the function *fdError()* can be called to determine the error:

EPFNOSUPPORT	The specified domain was not PF_INET.
EPROTOTYPE	The *type* parameter does not support the *protocol* parameter.
ESOCKTNOSUPPORT	The specified socket type is not supported.
ENOMEM	Memory allocation error allocating socket buffers.
EMFILE	The descriptor table is full.

Description

The *socket()* function creates a socket, an endpoint for communication and returns the socket in the form of a file descriptor.

The *domain* parameter specifies a communications domain within which communication will take place; this selects the protocol/address family that should be used. These families are defined in the include file INC\SOCKET.H. This will always be PF_INET (AF_INET) in this implementation.

The socket type parameter specifies the semantics of communication. Currently defined types are:

SOCK_STREAM	Provides sequenced, reliable, two-way connection based byte streams. An out-of-band data transmission mechanism is supported.
SOCK_STREAMNC	Identical to SOCK_STREAM except that received data is not coalesced into a receive holding buffer. This eliminates one or two receive data copies (depending on which *recv()* socket function is used), but has the potential of tying up multiple data packets. It should only be used when the socket is to receive data in large bursts. Out-of-band data is supported, but only when the traditional *recv()* socket calls are used.
SOCK_DGRAM	Supports datagrams - connectionless, unreliable messages of a fixed (typically small) maximum length.
SOCK_RAW	Similar to SOCK_DGRAM, only allows the use of any protocol that must be manually constructed in each datagram by the programmer.

The *protocol* parameter specifies a particular protocol to be used with the socket. In this implementation of the stack, SOCK_STREAM must use IPPROTO_TCP, SOCK_DGRAM must use IPPROTO_UDP, and SOCK_RAW is unrestricted. To remain compatible with the industry, this parameter can be set to NULL on SOCK_STREAM or SOCK_DGRAM.

3.4 Full Duplex Pipes Programming Interface

3.4.1 Synopsis

Although sockets can be used for inter-task communications, it is not the most efficient method. The stack provides a second data communications model called pipes, which allow for local connection oriented communications.

A pipe is a full duplex connection oriented file descriptor. When a pipe is created, both ends of the pipe are returned to the caller as file descriptors. One end of the pipe can then be passed to another task by first converting it to a file handle with the *fdGetFileHandle()* function.

Communication is performed using the standard file and sockets API functions. All the file descriptor functions are supported with pipes: *fdSelect()*, *fdClose()*, *fdError()*, and *fdGetFileHandle()*.

Also, socket functions *send()* and *recv()* write and read data through the pipe. Both functions also support the following standard sockets message flags when using pipes:

MSG_PEEK	Examine data but do not consume it.
MSG_DONTWAIT	Do not block on send/recv operation (by default, pipe operations always block).

Pipes are connection oriented, thus, when one end closes, the other end is altered by an error return from *send()* or *recv()*. It is therefore possible to make a blocking call on *recv()* without concern that the function will be deadlocked if the other end terminates the connection.

3.4.2 Pipe API Functions

Because pipes share file descriptor and IO functions with sockets, the only pipe oriented function is the creation of the connected pair.

pipe — ***Create a Full Duplex Pipe***

Syntax

int pipe(HANDLE *pfd1, HANDLE *pfd2);

Parameters

pfd1	Pointer to file descriptor to first end of pipe.
pfd2	Pointer to file descriptor to second end of pipe.

Return Value

Returns zero on success or -1 on error. A more detailed error code can be found by calling *fdError()*.

Description

Creates a pre-connected full duplex pipe. The returned file descriptors can be used with all the fd file descriptor functions, as well as the send() and recv() socket functions.

Pipes are connection oriented, so like TCP, a read or write call can return ENOTCONN when the connection is broken by one side or the other.

Note: Both file descriptors must be closed to correctly close down (and free) a pipe.

3.5 Internet Group Management Protocol (IGMP)

3.5.1 Synopsis

Internet Group Management Protocol (IGMP) is designed to help routers in routing IP multicast traffic. Each router can have multiple ports, and it is inefficient for the router to replicate every IP multicast packet out of each active port. Using the IGMP protocol, the multicast router is able to keep track of which IP multicast addresses need to be routed to each individual port. This allows the router to limit IP multicast transmission to only those ports that require the multicast traffic.

The IGMP protocol assumes a client/server relationship between endpoints. The IGMP server is run by the multicast router to get IP multicast information about all the client on each of its individual ports. The IGMP client is only concerned with communicating its own multicast requirements to the local IGMP server, so that it will get the IP multicast packets that it requires.

The NDK does not currently support IP multicast routing, so there is no need to use IGMP in server mode. However, the software does support IGMP client operation.

The IGMP client module indicates to the IGMP server which multicast IP addresses that the client needs to receive. The IGMP API will also maintain the Ethernet multicast MAC address list at the Ethernet driver level.

3.5.2 Function Overview

Application Functions:

IGMPJoinHostGroup()	Join an IP Multicast Host Group
IGMPLeaveHostGroup()	Leave an IP Multicast Host Group

3.5.3 API Functions

Note that these functions are application functions, and do not need to be called from within an *llEnter()/llExit()* pair. (Doing so will cause an error.)

IGMPJoinHostGroup *Join an IP Multicast Host Group*

Syntax uint IGMPJoinHostGroup(IPN IpAddr, uint IfIdx);

Parameters

IpAddr	IP address of the host group to join
IfIdx	Interface index of the interface on which to join the group

Return Value Returns 1 if it was able to join the group, otherwise 0.

Description This function is called by the application to join a multicast host group. On calling the function, the IGMP module will convert the supplied IP address to the corresponding Ethernet MAC multicast address, and add it to the Ethernet layer (if the device is Ethernet). It will then perform the necessary IGMP operations to join the multicast group on the specified interface.

The system allows a maximum of 32 groups over four interfaces to be active at any given time.

IGMPLeaveHostGroup *Leave an IP Multicast Host Group*

Syntax void IGMPLeaveHostGroup(IPN IpAddr, uint IfIdx);

Parameters

IpAddr	IP address of the host group to leave
IfIdx	Interface Index of the interface on which to leave the group

Return Value None.

Description This function is called by the application to leave a multicast host group. On calling the function, the IGMP module will remove the corresponding Ethernet multicast address from the specified device index (if the device is Ethernet).

Initialization and Configuration

This chapter discusses the initialization and configuration processes for the NDK.

Topic **Page**

4.1 Configuration Overview

One of the more tedious aspects of using the stack system is the initialization process. Normally, configuration is not a concern in programming because it is part of the underlying OS. However, the target platform for this software is the embedded processor space where there is little or no configuration done for you, and you must usually develop custom boot code as well.

The NDK includes three things to increase the ease of the configuration and boot process. First, there is a programming API for creating, walking, and editing a system configuration. Secondly, initialization software is provided (with source code) that boots the system using a configuration created with the configuration API, and completely initializes the stack environment. Lastly, a user-expandable standard configuration specification is defined.

4.2 Configuration Manager

4.2.1 Synopsis

The configuration manager is a collection of API functions to help you create and manipulate a configuration. The manager API is independent of the configuration specification.

The configuration is arranged as a database with a master key (called *Tag*) that defines the class of configuration item. A second key (called *Item*) determines the sub-item type in the *tag* class. For each *tag* and *item*, there can be multiple instances. Items can be further distinguished by their *instance* value.

The configuration is based on an active database. That is, any change to the database can cause an immediate reaction in the system. For example, if a route is added to the configuration, it is added to the system route table. If the route is then removed from the configuration, it is removed from the system route table.

To facilitate the active procession of configuration changes in a generic fashion, the configuration API allows the installation of service provider callback functions that are called to handle specific *tag* values in the configuration.

Configurations can be set active or inactive. When a configuration is active, any change to the configuration results in a change in the system. When a configuration is inactive, it behaves like a standard database. Part of the main initialization sequence is to make the system configuration active, and then inactive when shutting down.

Both the configurations and configuration entries are referenced by a generic handle. Configuration functions (named as *CfgXxx()*) take a configuration handle parameter, while configuration entry functions (name as *CfgEntryXxx()*) take a configuration entry handle parameter. These handles are not interchangeable.

Configuration entry handles are *referenced*. This means that each handle contains an internal reference count so that the handle is not destroyed by one task while another task expects it to stay valid. Functions that return a configuration entry handle supply a referenced handle in that its reference count has already been incremented for the caller. The caller can hold this handle indefinitely, but should dereference it when it is through. There are three calls that dereference a configuration entry handle. These are: *CfgRemoveEntry()*, *CfgGetNextEntry()*, and most simply *CfgEntryDeRef()*. See individual function descriptions for more information.

The PPP module in the stack library and several modules in the NETTOOLS library make use of a *default* configuration to store and search for data. The default configuration is accessed by passing in a NULL configuration handle to any function that takes the *hCfg* parameter (except *CfgFree()*). The default configuration is specified by calling *CfgSetDefault()*.

4.2.2 Function Overview

The configuration access functions (in functional order) are as follows:

Configuration Functions:

CfgNew()	Create a new configuration
CfgFree()	Destroy a configuration
CfgSetDefault()	Set default configuration
CfgGetDefault()	Get default configuration
CfgLoad()	Load configuration from a linear memory buffer
CfgSave()	Save configuration to a linear memory buffer
CfgSetExecuteOrder()	Set the tag initialization and shutdown order on execute
CfgExecute()	Make the configuration active or inactive
CfgSetService()	Sets service callback function for a particular tag
CfgAddEntry()	Add a configuration entry to a configuration
CfgRemoveEntry()	Remove entry from configuration
CfgGetEntryCnt()	Get the number of item instances for a tag/item pair
CfgGetEntry()	Get a referenced handle to a configuration entry
CfgGetNextEntry()	Return supplied entry handle and get next entry handle
CfgGetImmediate()	Get configuration entry data without getting an entry handle

Configuration Entry Functions:

CfgEntryRef()	Add a reference to a configuration entry handle
CfgEntryDeRef()	Remove a reference to a configuration entry handle
CfgEntryGetData()	Get configuration entry data from entry handle
CfgEntrySetData()	Replace data block of entry data using entry handle
CfgEntryInfo()	Get information on a configuration entry handle

4.2.3 Configuration API Functions

CfgAddEntry — *Add Configuration Entry to Configuration*

Syntax

int CfgAddEntry(HANDLE hCfg, uint Tag, uint Item, uint Mode, uint Size, UINT8 *pData, HANDLE *phCfgEntry);

Parameters

hCfg	Handle to configuration
Tag	Tag value of new entry
Item	Item value of new entry
Mode	Mode flags for how to add entry
Size	Size of entry data pointed to by pData
pData	Pointer to entry data
phCfgEntry	Pointer to where to write handle of new configuration entry

Return Value

Returns 1 on success with successful processing by a service callback function (see *CfgSetService()*)

Returns 0 on success with no processing performed by a service callback function

Returns less than 0 but greater than CFGERROR_SERVICE on a configuration error

The possible configuration errors are:

CFGERROR_BADHANDLE	Invalid hCfg handle
CFGERROR_BADPARAM	Invalid function parameter
CFGERROR_RESOURCES	Memory allocation error while adding entry

Returns less than or equal to CFGERROR_SERVICE when the service callback function returns an error. Service errors are specific to the service callback functions installed and are thus implementation dependent. The original error return from the service callback can be retrieved by using the CFG_GET_SERVICE_ERROR() macro:

```
ServiceErrorCode  = CFG_GET_SERVICE_ERROR(CfgAddEntryReturnValue);
```

Note: On a service error, the configuration entry is still added to the configuration, and an entry handle is written to phCfgEntry when the pointer is supplied.

Description

This function creates a new configuration entry and adds it to the configuration.

The *phCfgEntry* parameter is an optional pointer that can return a handle to the newly added configuration entry. When the *phCfgEntry* parameter is valid, the function writes the referenced handle of the new configuration entry to the location specified by this parameter. It is then the caller's responsibility to dereference this handle when it is finished with it. When the parameter *phCfgEntry* is NULL, no entry handle is returned, but the function return value is still valid.

Configuration entry handles are dereferenced by calling one of the following:

CfgEntryDeRef()	Stop using the entry
CfgRemoveEntry()	Stop using entry and remove it from the configuration
CfgGetNextEntry()	Stop using entry and get next entry

If the execution state of the configuration is active (see *CfgExecute()*), the addition of the configuration entry is immediately reflected in the operating state of the system.

Multiple configuration entries can exist with the same Tag and Item key values. The system creates a third key (Instance) to track these duplicate keyed entries. However, by default, the configuration system does not allow for fully duplicate entries. Entries are full duplicates if there exists another entry with the same Tag and Item key values and an exact duplicate data section (size and content). When a full duplicate entry is detected, the new (duplicate) entry is not created.

There are some options that determine how the entry is added to the configuration by using flags that can be set in the Mode parameter. The default behavior when adding an object is as follows:

- Multiple instances with the same *Tag* and *Item* values are allowed.
- However, duplicate instances with the same *Tag*, *Item*, *Size*, and *pData* contents are ignored.
- New entries are saved to the linear buffer if or when *CfgSave()* is used.

To modify the default behavior, one or more of the following flags can be set:

CFG_ADDMODE_UNIQUE	Replace all previous entry instances with this single entry.
CFG_ADDMODE_DUPLICATE	Allow full duplicate entry (duplicate Tag, Item, and entry data). Requests to add duplicates are normally ignored.
CFG_ADDMODE_NOSAVE	Do not include this entry in the linear buffer in *CfgSave()*.

Note: Setting both the CFG_ADDMODE_UNIQUE and CFG_ADDMODE_DUPLICATE flags is the same as only setting CFG_ADDMODE_UNIQUE.

CfgExecute — *Set the Execution State (Active/Inactive) of the Configuration*

Syntax

int CfgExecute(HANDLE hCfg, int fExecute);

Parameters

hCfg	Handle to configuration
fExecute	Desired execute state (1 = active)

Return Value

Returns 0 on success, or less than 0 on an operation error. The possible errors are:

CFGERROR_BADHANDLE	Invalid hCfg handle
CFGERROR_BADPARAM	Invalid function parameter
CFGERROR_ALREADY	Configuration is already in desired state

Description

When a configuration is first created, it is in an inactive state, so changes to the configuration are not reflected by changes to the system.

Executing the configuration (setting *fExecute* to 1) causes all current entries in the configuration to be loaded, and any further changes in the configuration to be immediately reflected in the system.

Disabling execution of the configuration (setting *fExecute* to 0) causes all configuration entries to be unloaded from the system (note that they are not removed from the

configuration). Any further changes to the configuration are not reflected by changes to the system.

CfgFree — *Destroy a Configuration Handle*

Syntax

void CfgFree(HANDLE hCfg);

Parameters

hCfg	Handle to configuration

Return Value

None.

Description

Destroys a configuration. Unloads and frees all configuration entries and frees the configuration handle. After this call, the configuration handle *hCfg* is invalid.

CfgGetDefault — *Get Default Configuration Handle*

Syntax

HANDLE CfgGetDefault();

Parameters

None.

Return Value

Returns a handle to the current default configuration, or NULL if None.

Description

This function returns the current default configuration handle. The default handle is used in any function that takes a *hCfg* parameter, when the specified parameter is NULL. At initialization, there is no default configuration. It must be allocated by CfgNew() and then specified via *CfgSetDefault()*. Normally, the default configuration is reserved for system use.

CfgGetEntry — *Get Configuration Entry from Configuration*

Syntax

int CfgGetEntry(HANDLE hCfg, uint Tag, uint Item, uint Index, HANDLE *phCfgEntry);

Parameters

hCfg	Handle to configuration
Tag	Tag value of entry
Item	Item value of entry
Index	Instance index to get (1 to *n*)
phCfgEntry	Pointer to where to write configuration entry handle

Return Value

Returns 1 if a matching entry was found

Returns 0 if a matching entry was not found

Returns less than 0 on error

The possible configuration errors are:

CFGERROR_BADHANDLE	Invalid *hCfg* handle
CFGERROR_BADPARAM	Invalid function parameter

Description

This function searches the configuration for an entry matching the supplied *Tag* and *Item* parameters and an index matching the supplied *Index* parameter. For example, when *Index* is 1, the first instance is returned, when *Index* is 2, the second instance is returned. The total number of instances can be found by calling *CfgGetEntryCnt()*.

The *phCfgEntry* parameter is an optional pointer that can return the handle of the configuration entry found by this function. When the *phCfgEntry* parameter is valid, the function writes the referenced handle of the configuration entry found to this pointer. It is

the caller's responsibility to dereference the handle when it is no longer needed. When the parameter *phCfgEntry* is NULL, no entry handle is returned, but the function return value is still valid (found or not found).

Configuration entry handles are dereferenced by the calling one of the following:

CfgEntryDeRef()	Stop using the entry
CfgRemoveEntry()	Stop using entry and remove it from the configuration
CfgGetNextEntry()	Stop using entry and get next entry

Note: Do not attempt to use the *Index* value to enumerate all entry instances in the configuration. The index of an entry handle is valid only at the time of the call as an item can move up and down in the list as configuration changes are made. To enumerate every entry for a Tag/Item pair, start with Index 1, and then use *CfgGetNextEntry()* to get additional entries.

CfgGetEntryCnt — *Get the Number of Entry Instances for the Supplied Tag/Item Pair*

Syntax

int CfgGetEntryCnt(HANDLE hCfg, uint Tag, uint Item);

Parameters

hCfg	Handle to configuration
Tag	Tag value of query
Item	Item value of query

Return Value

Returns 0 or greater on success (number if instances found) or less than 0 on error.

The possible errors are:

CFGERROR_BADHANDLE	Invalid hCfg handle
CFGERROR_BADPARAM	Invalid function parameter

Description

This function searches the configuration for all instances matching the supplied Tag and Item parameters and returns the number of instances found.

CfgGetImmediate — *Get Configuration Entry Data Directly from Configuration*

Syntax

int CfgGetImmediate(HANDLE hCfg, uint Tag, uint Item, uint Index, int Size, UINT8 *pData);

Parameters

hCfg	Handle to configuration
Tag	Tag value of entry
Item	Item value of entry
Index	Instance index to get (1 to *n*)
Size	Size of buffer to receive data
pData	Pointer to data buffer to receive data

Return Value Number of bytes copied

Description This function is a useful shortcut when searching the configuration for well known entries. It searches the configuration for entries matching the supplied *Tag* and *Item* parameters and uses the item matching the supplied Index parameter. For example, if *Index* is 1, the first instance is used, if *Index* is 2, the second instance is used. The total number of instances can be found by calling *CfgGetEntryCnt()*.

Instead of returning a referenced handle to the configuration entry (as with the more generic *CfgGetEntry()* function), this function immediately gets the entry data for this entry and copies it to the data buffer pointed to by *pData*.

The increased simplicity does decrease the function's flexibility. This function returns the number of bytes copied, so it will return 0 for any of the following reasons:

- A supplied parameter is incorrect
- The item was not found
- The supplied buffer size (specified by Size) was not large enough to hold the data

CfgGetNextEntry *Get the Next Entry Instance Matching the Supplied Entry Handle*

Syntax int CfgGetNextEntry(HANDLE hCfg, HANDLE hCfgEntry, HANDLE *phCfgEntryNext);

Parameters

hCfg	Handle to configuration
hCfgEntry	Handle to last configuration entry
phCfgEntryNext	Pointer to receive handle of next configuration entry

Return Value Returns 1 if a next entry was found

Returns 0 if a next entry was not found

Returns less than 0 on error

The possible configuration errors are:

CFGERROR_BADHANDLE	Invalid *hCfg* handle
CFGERROR_BADPARAM	Invalid function parameter

Note: The handle *hCfgEntry* is not dereferenced on the event of an error.

Description This function serves two purposes. First, it dereferences the configuration entry handle supplied in *hCfgEntry*. After this call, the handle is invalid (unless there was more than one reference to it). Secondly, this function returns a referenced configuration entry handle to the next instance (if any) of an entry that matches the Tag and Item values of the supplied entry.

When the parameter *phCfgEntryNext* is NULL, no entry handle is returned, but the function always returns 1 if such an entry was found and 0 when not.

When the *phCfgEntryNext* parameter is not NULL, the function writes a referenced handle to the configuration entry to the location specified by this parameter. It is then the caller's responsibility to dereference this handle when it is finished with it.

Configuration entry handles are dereferenced by the calling one of the following:

CfgEntryDeRef()	Stop using the entry
CfgRemoveEntry()	Stop using entry and remove it from the configuration
CfgGetNextEntry()	Stop using entry and get next entry

CfgLoad — *Load a Configuration from a Linear Memory Block*

Syntax

int CfgLoad(HANDLE hCfg, int Size, UINT8 *pData);

Parameters

hCfg	Handle to configuration
Size	Size of memory block to load
pData	Pointer to memory block to load

Return Value

Returns the number of bytes loaded, or less than 0 on an error. The possible errors are:

CFGERROR_BADHANDLE	Invalid *hCfg* handle
CFGERROR_BADPARAM	Invalid function parameter

Description

The configuration system features the ability for the manager to convert a configuration database to a linear block of memory for storage in non-volatile memory. The configuration can then be converted back on reboot.

This function converts a linear block of memory to a configuration by loading each configuration entry it finds in the coded data block. Note that *CfgLoad()* can be used to load entries into a configuration that already has pre-existing entries, but the method of entry is not preserved (see *Mode* parameter of *CfgAddEntry()*). To ensure that the resulting configuration exactly matches the one converted with *CfgSave()* , this function should only be called on an empty configuration handle.

CfgNew — *Create a New Configuration*

Syntax

HANDLE CfgNew();

Parameters

None.

Return Value

Returns handle to a new configuration or NULL on memory allocation error.

Description

Creates a configuration handle that can be used with other configuration functions. The new handle defaults to the inactive state (see *CfgExecute()*).

CfgRemoveEntry — *Remove Configuration Entry from Configuration by Handle*

Syntax

int CfgRemoveEntry(HANDLE hCfg, HANDLE hCfgEntry);

Parameters

hCfg	Handle to configuration
hCfgEntry	Configuration entry to remove

Return Value

Returns 0 on success or less than 0 on error.

The possible errors are:

CFGERROR_BADHANDLE	Invalid *hCfg* handle
CFGERROR_BADPARAM	Invalid function parameter

Note: The handle *hCfgEntry* is not dereferenced on the event of an error.

Description

This function removes a configuration entry from a configuration.

If the execution state of the configuration is active (see *CfgExecute()*), then the removal of the configuration entry is immediately reflected in the operating state of the system.

This function also performs a single dereference operation on the configuration entry handle, so the handle is invalid after the call (unless there was more than one reference made). Although the entry handle is not freed until all handle references have been removed, it is always removed from the configuration immediately.

CfgSave

Save a Configuration to a Linear Memory Block

Syntax

int CfgSave(HANDLE hCfg, int *pSize, UINT8 *pData);

Parameters

hCfg	Handle to configuration
pSize	Pointer to size of memory block
pData	Pointer to memory block to load

Return Value

Returns the number of bytes written, 0 on a size error (value at *pSize* set to required size), or less than 0 on an operation error. The possible errors are:

CFGERROR_BADHANDLE	Invalid *hCfg* handle
CFGERROR_BADPARAM	Invalid function parameter

Description

One of the features of the configuration system is the ability for the manager to convert a configuration database to a linear block of memory for storage in non-volatile memory. The configuration can then be converted back on reboot.

This function saves the contents of the configuration specified by hCfg into the linear block of memory pointed to by *pData*.

The size of the data buffer is initially pointed to by the *pSize* parameter. If this size value pointed to by this pointer is zero (*pSize* cannot itself be NULL), the function does not attempt to save the configuration but rather calculates the size required and writes this value to the location specified by *pSize*. In fact, any time the value at *pSize* is less than the size required to store the configuration, the function returns 0 and the value at *pSize* is set to the size required to store the data.

The *pData* parameter points to the data buffer to receive the configuration information. This pointer can be null if **pSize* is zero. Note that the pointer *pSize* must always be valid.

CfgSetDefault

Set Default Configuration Handle

Syntax

HANDLE CfgSetDefault(HANDLE hCfg);

Parameters

hCfg	Handle to configuration to set as default, or NULL to clear default

Return Value

None.

Description

This function sets the current default configuration handle to that specified in *hCfg*. The default handle is used in any function that takes a hCfg parameter, when the specified parameter is NULL. At initialization, there is no default configuration. It must be allocated by *CfgNew()* and then specified via *CfgSetDefault()*. Normally, the default configuration is reserved for system use. The default configuration handle should not be freed until it is

cleared by calling *CfgSetDefault(0)*.

CfgSetService *Set Service Callback Function for Configuration Tag*

Syntax

int CfgSetService(HANDLE hCfg, uint Tag, int (*pCb) (HANDLE, uint, uint, uint, HANDLE));

Parameters

hCfg	Handle to configuration
Tag	Tag value to change
pCb	Pointer to service callback function

Return Value

Returns 0 on success, or less than 0 on error. The possible errors are:

CFGERROR_BADHANDLE	Invalid *hCfg* handle
CFGERROR_BADPARAM	Invalid function parameter

Description

To give the configuration the ability to be active - i.e., to make real-time changes to the system as the configuration changes, the configuration manager must have the ability to make changes to the system. To enable this in a generic fashion, the configuration manager allows for the installation of service callback functions for each configuration *tag* value.

This function sets the service function for a particular configuration *tag*. Service function pointers default to NULL, and when they are NULL, no service is performed for the configuration entry (it becomes information data only).

When invoked, the service callback function is passed back information about the affected entry. The callback function is defined as:

int CbSrv(HANDLE hCfg, uint Tag, uint Item, uint Op, HANDLE hCfgEntry),

hCfg	HANDLE to Config
Tag	Tag value of entry changed
Item	Item value of entry changed
Op	Operation (CFGOP_ADD or CFGOP_REMOVE)
hCfgEntry	Non-Referenced HANDLE to entry added or removed

The callback should return 1 on success, 0 on pass, and <0 on error.

Note: The configuration entry handle passed to the callback function is not referenced, as its scope expires when the callback function returns.

CfgSetExecuteOrder *Set the Tag Initialization and Shutdown Order on Execute*

Syntax

int CfgSetExecuteOrder(HANDLE hCfg, uint Tags, uint *pOpenOrder, uint *pCloseOrder);

Parameters

hCfg	Handle to configuration
Tags	Number of tag values in *pOpenOrder* and *pCloseOrder*
pOpenOrder	Pointer to array of tag values in initialization order

pCloseOrder Pointer to array of tag values in shutdown order

Return Value Returns zero on success, or less than 0 on an operation error. The possible errors are:

CFGERROR_BADHANDLE Invalid *hCfg* handle

CFGERROR_BADPARAM Invalid function parameter

Description The configuration API has no knowledge of the configuration database specification. Thus, it has no concept of a priority in loading and unloading configuration entries. The default order for both loading and unloading is by ascending tag value.

You may require that the application specify the exact order in which entries should be initialized (specified in *pOpenOrder*) and shut down (specified in *pCloseOder*). Both arrays must be provided - even if they are identical pointers. The number of elements in each array is specified by the Tags parameter. This must exactly match the maximum number of tags in the system defined by CFGTAG_MAX. An entry of 0 in either order array is used as a placeholder for tags that have not yet been defined.

4.2.4 Configuration Entry API Functions

CfgEntryDeRef — *Remove a Reference to a Configuration Entry Handle*

Syntax int CfgEntryDeRef(HANDLE hCfgEntry);

Parameters

hCfgEntry Handle to configuration entry

Return Value Returns 0 on success, or less than 0 on an operation error. The possible errors are:

CFGERROR_BADHANDLE Invalid *hCfgEntry* handle

Description This function removes a reference to the configuration entry handle supplied in hCfgEntry. It is called by an application when it wishes to discard a referenced configuration entry handle. Once this function is called, the handle should no longer be used.

CfgEntryGetData *Get Configuration Entry Data*

Syntax

int CfgEntryGetData(HANDLE hCfgEntry, int *pSize, UINT8 *pData);

Parameters

hCfgEntry	Handle to configuration entry
pSize	Pointer to size of data buffer
pData	Pointer to data buffer

Return Value

Returns the number of bytes written, 0 on a size error (value at *pSize* set to required size), or less than 0 on an operation error. The possible errors are

CFGERROR_BADHANDLE	Invalid *hCfgEntry* handle
CFGERROR_BADPARAM	Invalid function parameter

Description

This function acquires the entry data of the configuration entry specified by the entry handle in *hCfgEntry*.

The value pointed to by *pSize* is set to the size of the supplied buffer, or zero to get the required size (the pointer *pSize* must be valid, but the value at the pointer can be zero). If the value at *pSize* is zero, or less than the number of bytes required to hold the entry data, this function returns 0, and the number of bytes required to hold the data is stored at *pSize*.

The *pData* parameter points to the data buffer to receive the configuration entry data. This pointer can be null if **pSize* is zero.

CfgEntryInfo *Get Information on a Configuration Entry*

Syntax

int CfgEntryInfo(HANDLE hCfgEntry, int *pSize, UINT8 **ppData);

Parameters

hCfgEntry	Handle to configuration entry
pSize	Location to receive the size of the configuration entry data buffer
ppData	Location to receive the pointer to the configuration entry data buffer

Return Value

Returns 0 on success, or less than 0 on an operation error. The possible errors are:

CFGERROR_BADHANDLE	Invalid *hCfgEntry* handle

Description

This function acquires the size and pointer to a configuration entry's data buffer.

The entry handle is supplied *hCfgEntry*. A pointer to receive the size of the entry's data buffer is supplied in *pSize*, and a pointer to receive a pointer to the entry's data buffer is supplied in *ppData*. Either pointer parameter can be left NULL if the information is not required.

This function should be used with great care. Direct manipulation of the configuration entry data should only be attempted on informational tags, and only when the caller holds a referenced handle to the configuration entry. This function is used in configuration service callback functions, which are called only when the configuration is in a protected state.

CfgEntryRef *Add a Reference to a Configuration Entry Handle*

Syntax

int CfgEntryRef(HANDLE hCfgEntry);

Parameters

hCfgEntry	Handle to configuration entry

Return Value Returns 0 on success, or less than 0 on an operation error. The possible errors are:

CFGERROR_BADHANDLE	Invalid *hCfgEntry* handle

Description This function adds a reference to the configuration entry handle supplied in *hCfgEntry*. It is called by an application when it intends to use a configuration entry handle beyond the scope of the function that obtained it from the configuration. This normally occurs when one user function calls another and passes it a handle.

The handle should be dereferenced when no longer needed. Configuration entry handles are dereferenced by calling one of the following:

CfgEntryDeRef()	Stop using the entry
CfgRemoveEntry()	Stop using entry and remove it from the configuration
CfgGetNextEntry()	Stop using entry and get next entry

CfgEntrySetData *(Re)Set Configuration Entry Data*

Syntax int CfgEntrySetData(HANDLE hCfgEntry, int Size, UINT8 *pData);

Parameters

hCfgEntry	Handle to configuration entry
Size	Size of data buffer
pData	Pointer to data buffer

Return Value Returns the number of bytes written, 0 on a size error (new size does not match old size), or less than 0 on an operation error. The possible errors are:

CFGERROR_BADHANDLE	Invalid *hCfgEntry* handle
CFGERROR_BADPARAM	Invalid function parameter

Description This function replaces the entry data of the configuration entry specified by the entry handle in *hCfgEntry*.

The new entry data is pointed to by the *pData* parameter, with a size indicated by *Size*. Note that the new data must be an exact replacement for the old. The size of the new buffer must exactly match the old size.

This function should be used for configuration entries that are for information purposes only. Note that if a service provider callback is associated with the Tag value of this entry, the processing function is not called as a result of this data update. This function only updates the data stored for this configuration entry.

4.3 Network Control Initialization Procedure (NETCTRL)

4.3.1 Synopsis

As previously mentioned, the configuration and initialization of the stack is tedious. For this reason, the stack library includes code to perform system initialization based on a configuration constructed by the programmer. This configuration can be manually built through the configuration API, or you can build it with a configuration utility (not included in the stack).

The basic initialization of the scheduling routines is performed by a network control layer called NETCTRL. The source code to this layer is user-serviceable. See the *TMS320C6000 Network Developer's Kit (NDK) Software User's Guide* (SPRU523) for more details.

4.3.2 Basics

The basic process of stack initialization is as follows:

1. Initialize the operating system environment with the initialization function *NC_SystemOpen(Priority, OpMode)*. This function must always be called first - before any other NDK related function. The calling parameters determine the priority and operating mode of the network event scheduler.
2. Create a new configuration via *CfgNew()*.
3. Build the new configuration via configuration API calls, or load a previous configuration from non-volatile memory using *CfgLoad()*.
4. Boot the stack with the configuration by calling *NC_NetStart(hCfg, pfnStart, pfnStop, pfnNetIP)* with a handle to the configuration, plus pointers to three user supplied callback functions for start, stop, and IP address change operations. The *NC_NetStart()* function does not return until the stack session has terminated. The configuration handle *hCfg* becomes the default configuration for the system.
5. After some preliminary initialization, the *NC_NetStart()* function creates a new thread that calls the user supplied callback function for the start operation. At this point, the callback function creates task threads for its networking requirements. This start function does not need to return immediately, but should return at some point - i.e., the callback function should not take permanent control of the calling thread. If system shutdown is initiated before the start function returns, some resources may not be freed.
6. Under normal operation, the network does not shut down until the *NC_NetStop()* function is called. At some point after a call to *NC_NetStop()*, the original *NC_NetStart()* thread calls the user supplied callback function for the stop operation. In this callback function, the application shuts down any operation it initiated in the start callback function and frees any allocated resources. After the stop callback function returns, NDK functionality is no longer available.
7. The original call to *NC_NetStart()* returns with the return value as set by the return parameter passed in the call to *NC_NetStop()*. The application can immediately reboot the NDK by calling *NC_NetStart()* again, with or without reloading a new configuration. This is useful for a reboot command.

When the system is ready for a final shutdown, the following actions are performed:

1. When *NC_NetStart()* returns and the session is over, call the *CfgFree()* function to free the configuration handle created with *CfgNew()*.
2. After all resources have been freed, call the *NC_SystemClose()* function to complete the system shutdown.

4.3.3 Function Overview

The system initialization access functions (in functional order) are as follows:

NC_SytemOpen()	Initiate a system session
NC_SystemClose()	Full system shutdown
NC_NetStart()	Start the network with a supplied configuration
NC_NetStop()	Halt the network, and pass a return code the caller of the *NC_NetStart()* function

4.3.4 Network Control API Functions

NC_SystemOpen — *Initiate a System Session*

Syntax

int NC_SystemOpen(int Priority, int OpMode);

Parameters

Priority	Network event scheduler task priority
OpMode	Network event scheduler operating mode

Return Value Returns the status of the open call, zero on success, or one of the following on error.

NC_OPEN_ILLEGAL_PRIORITY	*Priority* is not set to NC_PRIORITY_LOW or NC_PRIORITY_HIGH.
NC_OPEN_ILLEGAL_OPMODE	*OpMode* is not set to NC_OPMODE_POLLING or NC_OPMODE_INTERRUPT. Or, attempt to combine NC_OPMODE_POLLING with NC_PRIORITY_HIGH.
NC_OPEN_MEMINIT_FAILED	Memory initialization failed.
NC_OPEN_EVENTINIT_FAILED	Event initialization failed.

Description This is the first function that should be called when using the stack. It initializes the stack's memory manager, and the OS (or OS adaptation layer). It also configures the network event scheduler's task priority and operating mode.

Priority is set to either NC_PRIORITY_LOW or NC_PRIORITY_HIGH, and determines the scheduler task's priority relative to other networking tasks in the system.

OpMode is set to either NC_OPMODE_POLLING or NC_OPMODE_INTERRUPT, and determines when the scheduler attempts to execute. The interrupt mode is used in the vast majority of applications.

Note that polling operating mode attempts to run continuously, so when polling is used, Priority must be set to NC_PRIORITY_LOW.

NC_SystemClose *Shutdown the System*

Syntax void NC_SystemClose();

Parameters None.

Return Value None.

Description This is the last function that should be called when using the stack. It shuts down the memory manager and performs a final memory analysis.

NC_NetStart *Start Network*

Syntax int NC_NetStart(HANDLE hCfg, void (*NetStartCb)(), void (*NetStopCb)(), void (*NetIPCb)(IPN,uint,uint));

Parameters

hCfg	Handle to network configuration
NetStartCb	Pointer to callback function called when network is started
NetStopCb	Pointer to callback function called when network is stopped
NetIPCb	Pointer to callback function called when an IP address is added or removed from the system

Return Value Returns the integer value passed to *NC_NetStop()*.

Description This function is called to boot up the network using the network configuration supplied in hCfg. Along with the network configuration, three callback function pointers are provided. These callback functions are called at distinct times. *NetStartCb()* is called when the system is first ready for the creation of application supplied network tasks, *NetStopCb()* is called when the network is about to shut down, and *NetIPCb()* is called when an IP address is added or removed from the system. If any of these callback functions are not

required, the function pointers can be set to NULL.

The *NC_NetStart()* function will not return until the entire network session has completed. Thus, all user supplied network code (creation of user tasks) should be included in the *NetStartCb()* function.

When *NetStartCb()* is called, the configuration handle supplied in *hCfg* is the default configuration handle for the system. The execution thread on which *NetStartCb()* is called is not critical to event scheduling, but it should return eventually; i.e., the application should not take control of the thread. If system shutdown is initiated before this callback function returns, some resources may not be freed.

Excluding critical errors, *NC_NetStart()* will return only if an application calls the *NC_NetStop()* function. The parameter passed to *NC_NetStop()* becomes the return value returned by *NC_NetStart()*.

Sometime after *NC_NetStop()* is called, but before *NC_NetStart()* returns, the *NC_NetStart()* thread will make a call to the application's *NetStopCb()* callback function. In this callback function, the application should shut down any task initiated in its *NetStartCb()* callback.

When an IP addressing change is made to the system, the *NetIPCb()* function is called. The callback function is declared as:

```
void NetIPCb( IPN IPAddr, uint IfIndex, uint fAdd );
```

IPAddr	IP Address being added or removed
IfIndex	Index of physical interface gaining or losing the IP address
fAdd	Set to 1 when adding an address, or 0 when removing an address

The *NetIPCb()* callback is purely informational, and no processing is necessary on the information provided.

There is an option for immediately calling *NC_NetStart()* again upon return, which provides a good stack reboot function. Optionally, the configuration can also be reloaded, which allows the stack to be restarted after a major configuration change.

NC_NetStop — *Stop Network*

Syntax

void NC_NetStop(int StopCode);

Parameters

StopCode	Return code to be returned by *NC_NetStart()*.

Return Value

None.

Description

This function is called to shut down a network initiated with *NC_NetStart()*. The return value supplied in the StopCode parameter becomes the return value for *NC_NetStart()*. See the description of *NC_NetStart()* for more detail.

4.4 Configuration Specification

4.4.1 Synopsis

The specification of all the various configuration options for the stack would require a separate document. This section details that part of the configuration that is relied upon by the Network Control (NC) initialization functions, or the services contained in the NETTOOLS library. The stack itself does not reference the configuration system. It has its own simpler method that is detailed in the appendix, but it is redundant when using the configuration API. In fact, they conflict, as the Network Control functions assume full control of it.

4.4.2 Organization

As already mentioned, the configuration is arranged as a database with the value Tag as a major key, and the value Item as a minor key. Every major stack configuration component has a major key (Tag) value, including: network services (protocol servers), connected IP networks, gateway routes, connected client entities, global system information, and low-level stack configuration.

Most of these tags require service callback functions to implement the system functionality. For example, when an IP network is added using the CFGTAG_IPNET tag, there must be a function that makes the corresponding system calls that adds the network to the system route table. All these server callback functions are contained in the NETCTRL directory. Although source code to these functions is provided, many of the system calls they make can only be understood by reading the attached appendices.

The tag values currently defined are:

CFGTAG_SERVICE	Network Service
CFGTAG_IPNET	IP Network (Address, subnet mask, etc.)
CFGTAG_ROUTE	IP Gateway Route
CFGTAG_CLIENT	IP Client (Client IP, Hostname, etc)
CFGTAG_ACCT	Client user account (name, password, etc.)
CFGTAG_SYSINFO	Global System Information
CFGTAG_OS	Operating System Configuration entry
CFGTAG_IP	IP Stack Configuration entry

4.4.3 Network Service Specification (CFGTAG_SERVICE)

The network services tag is perhaps the most time saving feature of the configuration. It allows you to instruct the system of what tasks to execute, and how they should be executed. It is also the most complicated configuration entry.

Network services are identified by a configuration *Tag* parameter value of CFGTAG_SERVICE.

Note that all these services are obtained directly from the NETTOOLS services API. The configuration system adds a level of abstraction so that a list of services can be added to a configuration, and then the service provider callback functions contained in the Network Control initialization routines can automatically load the services at runtime without having to call the NETTOOLS API directly.

4.4.3.1 Service Types

The type of service is indicated by the value of the *Item* parameter supplied to the *CfgAddEntry()* function. The defined service types include (by *Item*):

CFGITEM_SERVICE_TELNET	Telnet Server
CFGITEM_SERVICE_HTTP	HTTP Server
CFGITEM_SERVICE_NAT	Network Address Translation System
CFGITEM_SERVICE_DHCPSERVER	DHCP Server
CFGITEM_SERVICE_DHCPCLIENT	DHCP Client
CFGITEM_SERVICE_DNSSERVER	DNS Server

4.4.3.2 Common Argument Structure

Each individual service has its own specific configuration instance structure, but they all share a generic argument structure. This is defined as follows:

```
// Common Service Arguments
typedef struct _ci_srvargs{
    uint    Item;                           // Copy of Item value (init to NULL)
    HANDLE  hService;                       // Handle to service (init to NULL)
    uint    Mode;                           // Flags
    uint    Status;                         // Service Status (init to NULL)
    uint    ReportCode;                     // Standard NETTOOLS Report Code
    uint    IfIdx;                          // If physical Index
    IPN     IPAddr;                         // Host IP Address
    void(*pCbSrv)(uint, uint, uint, HANDLE); // CbFun for status change
    } CISARGS;
```

The individual fields are defined as follows:

- `uint Item;`
  This is a copy of the Item value used when the entry is added to the configuration. Its initial value should be NULL, but it is overwritten by the service provider callback. It is used so that the status callback function can be provided with the original Item value.
- `HANDLE hService;`
  This is the handle to the service as returned by the NETTOOLS function corresponding to the type of service requested. Its initial value should be NULL, and it is initialized by the service callback function when the service is started. The value is needed to shut down the service when the configuration is unloaded.
- `uint Mode;`
  The mode parameter is a collection of flags representing the desired execution behavior of the service. One or more of the following flags can be set:

CIS_FLG_IFIDXVALID	Specifies the *IfIdx* field is valid.
CIS_FLG_RESOLVEIP	Requests that *IfIdx* be resolved to an IP address before service execution is initiated.
CIS_FLG_CALLBYIP	Specifies that the service should be invoked by IP address. (*This is the default behavior when IFIDXVALID is not set, but this flag can be set with IFIDXVALID when RESOLVEIP is also set. If IFIDXVALID is set and this bit is not set, the service is invoked by physical device* .)
CIS_FLG_RESTARTIPTERM	A service that is dependent on a valid IP address (as determined by the RESOLVEIP flag) is shut down if the IP address becomes invalid. When this flag is set, the service will be restarted when a new address becomes available. Otherwise; the service will not be restarted.

- `uint Status;`
  The status parameter contains the service status as detected by the Net Control service callback function that initiates the service with NETTOOLS. The value of status should be initialized to NULL. Its defined values are:

CIS_SRV_STATUS_DISABLED	Service not active (NULL state)
CIS_SRV_STATUS_WAIT	Net Control is waiting on IP resolution to start service
CIS_SRV_STATUS_IPTERM	Service was terminated because it lost its IP address
CIS_SRV_STATUS_FAILED	Service failed to initialize via its NETTOOLS open function
CIS_SRV_STATUS_ENABLED	Service enabled and initialized properly

- `uint ReportCode;`
  All the services available via the configuration can also be launched directly via a NETTOOLS API. The NETTOOLS service API has a standard service reporting callback function that is mirrored by the configuration system via the Net Control service provider callback. This variable holds the last report code reported by the NETTOOLS service invoked by this configuration entry.
- `uint IfIdx;`
  This is the physical device Index (1 to n) on which the service is to be executed. For example, when launching a DHCP server service, the physical interface is that connected to the home network. For more generic services (like Telnet), the service can be launched by a pre-defined IP address (or INADDR_ANY as a wildcard). When launching by IP address only, this field is left NULL. If the field is valid, the CIS_FLG_IFIDXVALID flag should be set in *Mode*.
- `IPN IPAddr;`
  This is the IP address (in network format) on which to initiate the service. This IP address can specify the wildcard INADDR_ANY, in which case the service will accept connections to any valid IP address on any device. Note that some services (like DHCP server) do not support being launched by an IP address and require a device Index (supplied in *IfIdx*) on which to execute.
- `void(*pCbSrv)(uint, uint, uint, HANDLE);`
  The pCbSrv parameter contains a callback function that is called when the status of the service changes. It can be set to NULL if a callback is not required. The specification of the callback function is as follows:
- `void StatusCallback( uint Item, uint Status, uint Code, HANDLE hCfgEntry )`

Item	Item value of entry changed
Status	New status
Code	Report code (if any)
hCfgEntry	Non-Referenced HANDLE to entry with status change

Note that the *Status* parameter is the same as the *Status* field described in the CISARGS structure. The Code parameter is that returned by the NETTOOLS service callback, which is a lower-level status callback function used by Net Control.

4.4.3.3 Individual Configuration Entry Instance Structures

The following code defines the instance structures used for each of the defined configuration entries using the configuration service tag. Note that all structures contain the previously mentioned CISARGS structure. Some services require more information and their configuration entry structure contains an additional parameter structure as defined in the service's NETTOOLS API. Others do not require a parameter structure.

```
// Telnet Entry Data
typedef struct _ci_service_telnet {
    CISARGS        cisargs;         // Common arguments
    NTPARAM_TELNET param;           // Telnet parameters
    } CI_SERVICE_TELNET;

// HTTP Server Entry Data
```

```
typedef struct _ci_service_http {
    CISARGS cisargs;                // Common arguments
    NTPARAM_HTTP param;             // HTTP parameters
    } CI_SERVICE_HTTP;

// NAT Service Entry Data
typedef struct _ci_service_nat {
    CISARGS         cisargs;        // Common arguments
    NTPARAM_NAT     param;          // NAT parameters
    } CI_SERVICE_NAT;

// DHCP Server Entry Data
typedef struct _ci_service_dhcps {
    CISARGS         cisargs;        // Common arguments
    NTPARAM_DHCPS   param;          // DHCPS parameters
    } CI_SERVICE_DHCPS;

// DHCP Client Service
typedef struct _ci_service_dhcpc {
    CISARGS         cisargs;        // Common arguments
    NTPARAM_DHCP    param;          // DHCP parameters
    } CI_SERVICE_DHCPC;

// DNS Server Service
typedef struct _ci_service_dnss {
    CISARGS         cisargs;        // Common arguments
    } CI_SERVICE_DNSSERVER;
```

4.4.3.4 Specifying Network Services

For examples of adding specific network services to the configuration, please reference the service description in Chapter 6, *Network Tools Library Services*.

4.4.4 IP Network Specification (CFGTAG_IPNET)

The IPNET entry specifies what IP networks are to appear on which physical interfaces. When specifying an IPNET entry to the configuration, the Tag parameter is set to CFGTAG_IPNET, and the Item parameter is set to the Index (1 to n) of the physical interface on which the network is to appear.

The IPNET entry instance structure is defined as follows:

```
// IPNet Instance
typedef struct _ci_ipnet {
    uint    NetType;                // Network address type flags
    IPN     IPAddr;                 // IP Address
    IPN     IPMask;                 // Subnet Mask
    HANDLE  hBind;                  // Binding handle (initially NULL)
    char    Domain[CFG_DOMAIN_MAX]; // IPNet Domain Name
    } CI_IPNET;
```

The individual fields are defined as follows:

- `uint  NetType;`

CFG_NETTYPE_DYNAMIC	Address created by DHCP CLIENT
CFG_NETTYPE_VIRTUAL	Virtual Network used by DNS resolver
CFG_NETTYPE_DHCPS	Virtual Net Server reported by DHCP SERVER

This is type of network that appears on the interface. The network type determines how the network is treated by some services like NAT, DHCP, and DNS. The value is a collection of one or more of the following flags.

Most of the flags deal with the virtual network (or home network). If none of these flags are set, the network is a normal physical network. Note that virtual and non-virtual networks should not appear on the same interface. Also, only one network entry on each interface can have any of these flags set, although more than one of these flags can be set in that one entry.

- `IPN IPAddr;`

  This is the IP address of the stack on the designated interface. When the NetType flag DHCPS is set, this address is also the gateway address reported to DHCP clients served by the DHCP server service.
- `IPN IPMask;`

This is the IP network subnet mask.

- `HANDLE hBind;`

  This is the stack's internal binding handle for the network. Each connected network is represented as a binding internally to the stack. This is discussed further in the appendices at the end of this document. The value should be initialized to NULL.
- `char Domain[CFG_DOMAIN_MAX];`

  This is the domain name of the network. It should be a full domain like home1.net, not just home1.

4.4.5 IP Gateway Route Specification (CFGTAG_ROUTE)

The ROUTE entry specifies a route from one network to another via a specified IP gateway. When specifying a ROUTE entry to the configuration, the *Tag* parameter is set to CFGTAG_ROUTE, and the *Item* parameter is not used (set to zero).

The ROUTE entry instance structure is defined as follows:

```
// Route Instance
typedef struct _ci_route {
    IPN     IPDestAddr;             // Destination Network Address
    IPN     IPDestMask;             // Subnet Mask of Destination
    IPN     IPGateAddr;             // Gateway IP Address
    HANDLE  hRoute;                 // Route handle (initially NULL)
    } CI_ROUTE;
```

The individual fields are defined as follows:

- `IPN IPDestAddr;`

  This is the IP base address of the IP network of the network that is made accessible via the IP gateway. This value should be pre-masked with the *IPDestMask* so that:

  (IPDestAddr & IPDestMask) = IPDestMask

  This is used as a sanity check by the system. For a default route, the value is zero.
- `IPN IPDestMask;`

  This is the mask of the IP network accessible by the IP gateway. For a host route, the value is 0xFFFFFFFF, while for a default route, the value is zero.
- `IPN IPGateAddr;`

  This the IP address of the gateway through which the specified IP network is accessible. It must be an IP address that is available on a locally connected network, i.e., one gateway cannot point to another.
- `HANDLE hRoute;`

  This is a handle to the route created by this configuration entry. All routes are represented as route handles internally to the stack. This is discussed further in the appendices at the end of this document. The value should be initialized to NULL.

4.4.6 Client Record Specification (CFGTAG_CLIENT)

The CLIENT entry specifies a record of a client that appears on the indicated physical interface. When specifying a CLIENT entry to the configuration, the *Tag* parameter is set to CFGTAG_CLIENT, and the *Item* parameter is set to the index (1 to n) of the physical interface on which the client appears.

Client records exist for two purposes:

1. They are used to resolve DNS queries on virtual networks.
2. They are used by the DHCP server service to track DHCP clients on the serviced virtual network.

Client records are created automatically in some DHCP server configurations (when using an address pool), but they can also be added manually. This allows an application to build a pre-defined fixed list of clients and their designated IP addresses on a virtual (home) network.

The CLIENT entry instance structure is defined as follows:

```
typedef struct _ci_client {
    uint    ClientType;                // Entry Status
    uint    Status;                    // DHCPS Status (init to ZERO)
    IPN     IPAddr;                    // Client IP Address
    char    MacAddr[6];                // Client Physical Address
    char    Hostname[CFG_HOSTNAME_MAX]; // Client Hostname
    UINT32  TimeStatus;                // Time of last status change
    UINT32  TimeExpire;                // Expiration Time from TimeStatus
    } CI_CLIENT;
```

The individual fields are defined as follows:

- `uint  ClientType;`
  This is type of client record. There are only two types - those created by DHCP server from an address pool, and those created manually by an application.

CFG_CLIENTTYPE_DYNAMIC	Entry created via DHCPS
CFG_CLIENTTYPE_STATIC	Entry created manually

- `uint  Status;`
  This is status of the client record. It is used by the DHCP server to track the state of the client and its lease to its IP address. The status can also be NULL for STATIC entries.

CFG_CLIENTSTATUS_PENDING	Supplied via DHCP OFFER
CFG_CLIENTSTATUS_VALID	Validated by DHCP REQUEST
CFG_CLIENTSTATUS_STATIC	Reported via DHCP INFORM or non-DHCP application
CFG_CLIENTSTATUS_INVALID	Invalidated by DHCP DECLINE

- `IPN  IPAddr;`
  This is IP address of the client.
- `char  MacAddr[6];`
  This is physical Ethernet address of the client.
- `char  Hostname[CFG_HOSTNAME_MAX];`
  This is the hostname of the client. It is recorded by the DHCP server service, even if the record is STATIC. Thus, when running DHCP server, even with a fixed client list, DHCP clients can specify their own host names, and these names will be available to the DNS resolver, i.e., DNS server and DNS client.
- `UINT32  TimeStatus;`
  This is the last time that the *Status* parameter was validated. It is thus the start time of a DHCP client lease.
- `UINT32  TimeExpire;`
  This is the total time in seconds of a DHCP client lease reported by the DHCP server to its clients. When using an address pool for the DHCP server, the server chooses this value.

4.4.7 Client User Account (CFGTAG_ACCT)

The ACCT entry specifies an account record of a client that has access to the system. When specifying a ACCT entry to the configuration, the Tag parameter is set to CFGTAG_ACCT, and the Item parameter is set to the account type. Currently, the NDK has only one generic account type. Both PPP authentication and EFS authorization realms use this type. Valid types values are:

CFGITEM_ACCT_SYSTEM	System user account (PPP and EFS)
CFGITEM_ACCT_PPP	PPP user account (SYSTEM)
CFGITEM_ACCT_REALM	EFS Authorization Realm user account (SYSTEM)

The ACCT entry instance structure is defined as follows:

```
typedef struct _ci_acct {
    uint    Flags;                    // Account Flags
    char    Username[CFG_ACCTSTR_MAX]; // Username
    char    Password[CFG_ACCTSTR_MAX]; // Password
    } CI_ACCT;
```

The individual fields are defined as follows:

- `uint Flags;`
  The flags determine the access granted by channel or group. The channels or groups that any given PPP server will allow is determined when the PPP server is invoked. The same is true of the HTTP authentication realms. A single client account can be a member of one or more groups, therefore, one or more of the following flags can be set:

CFG_ACCTFLG_CH1	Allow access to channel/group/realm 1
CFG_ACCTFLG_CH2	Allow access to channel/group/realm 2
CFG_ACCTFLG_CH3	Allow access to channel/group/realm 3
CFG_ACCTFLG_CH4	Allow access to channel/group/realm 4

- `char Username[CFG_ACCTSTR_MAX];`
  This is the username of the client.
- `char Password[CFG_ACCTSTR_MAX];`
  This is the password corresponding to the supplied client username.

4.4.8 System Information Specification (CFGTAG_SYSINFO)

The SYSINFO entry contains various types of global system information. There is no service callback function associated with these entries, as they are static information only. When specifying a SYSINFO entry to the configuration, the *Tag* parameter is set to CFGTAG_SYSINFO, and the *Item* parameter is set to the system information item in question.

Note that the first 256 values for *Item* are reserved for items that exactly match the corresponding DHCP protocol information tag value. For example:

```
#define CFGITEM_DHCP_DOMAINNAMESERVER    6       // Stack's DNS servers
#define CFGITEM_DHCP_HOSTNAME            12      // Stack's host name
```

These values are read by various network services, and are written in one of two ways.

First, when the standard DHCP client is executing, it will take full control over the first 256 *Item* values. It fills in the entries when it obtains its address lease, and purges them when the lease expires. There is a set of default entries that the DHCP client will always request. Additional information requests can be made by configuring the DHCP client, and the resulting replies will be added to the configuration.

Second, when there is no DHCP client service, the network application must manually write values to the configuration for the *Item* values it views as important. A minimum configuration would include hostname, domain name, and a list of domain name servers. Note that multiple IP addresses should be stored as multiple instances of the same Item, not concatenated together with a longer byte length.

4.4.9 Extended System Information Tags

The following tag values are reserved for NDK and services configuration (see Appendix and Section E.3 for more information on PPP and HTTP realms):

CFGITEM_SYSINFO_REALM1	Realm Name 1 (maximum 31 chars)
CFGITEM_SYSINFO_REALM2	Realm Name 2 (maximum 31 chars)
CFGITEM_SYSINFO_REALM3	Realm Name 3 (maximum 31 chars)
CFGITEM_SYSINFO_REALM4	Realm Name 4 (maximum 31 chars)
CFGITEM_SYSINFO_REALMPPP	Server Name for PPP (maximum 31 chars)
CFGITEM_SYSINFO_EVALCALLBACK	Callback function registered by application. It is used by the Evaluation version of the NDK to notify the application five minutes before the expiration of the 24-hour evaluation period.

4.4.10 OS / IP Stack Configuration Item Specification (CFGTAG_OS, CFGTAG_IP)

The OS and IP tags specify entries that alter various configuration options that can be adjusted in the operating system and low-level stack operation. When specifying an entry to the configuration, the *Tag* parameter is set to CFGTAG_OS or CFGTAG_IP, and the *Item* parameter is set to the configuration item to set (these are listed below).

Creating a configuration entry results in an alteration of the system's internal configuration structures, but because these entries are also part of the configuration object (hCfg), they can be stored off and recorded as part of the CfgSave() functionality. Thus, using the configuration API has a significant advantage over modifying the internal structures manually.

Removing an entry restores the default value to the internal stack configuration. Entries that are not present cannot be read, and an error return on read implies the entry is in its default state.

The following is the list of configuration items. All items are of type int or uint. They correspond exactly to the internal system configuration structures. For more information on these fields, see the internal configuration discussion in both the Section 2.1.2 section earlier in this document, and the Configuring the Stack section in the attached appendix Section A.12.

When creating a configuration entry for one of these tags, the entry should be specified as unique. For example, to enable routing in the IP stack that code would be as follows:

```
// Enable IP routing
uint tmp = 1;
CfgAddEntry(hCfg, CFGTAG_IP, CFGITEM_IP_IPFORWARDING,
            CFG_ADDMODE_UNIQUE, sizeof(uint), (UINT8 *)&tmp, 0);
```

The following item values correspond directly to the OS and IP Stack configuration structures `_oscfg` and `_ipcfg`
For more information on these structures, see Section 2.1.2 and Section A.12.2.

When *Tag* is CFGTAG_OS, the value of *Item* can be one of the following:

CFGITEM_OS_DBGPRINTLEVEL	Debug message print threshold
CFGITEM_OS_DBGABORTLEVEL	Debug message abort threshold
CFGITEM_OS_TASKPRILOW	Lowest priority for stack task

CFGITEM_OS_TASKPRINORM	Normal priority for stack task
CFGITEM_OS_TASKPRIHIGH	Highest priority for stack task
CFGITEM_OS_TASKPRIKERN	Kernel-level priority (highest)
CFGITEM_OS_TASKSTKLOW	Minimum stack size
CFGITEM_OS_TASKSTKNORM	Normal stack size
CFGITEM_OS_TASKSTKHIGH	Stack size for high volume tasks

When *Tag* is CFGTAG_IP, the value of *Item* can be one of the following:

CFGITEM_IP_ICMPDOREDIRECT	Add route on ICMP redirect (1 = Yes)
CFGITEM_IP_ICMPTTL	TTL for ICMP messages (RFC1700 says 64)
CFGITEM_IP_ICMPTTLECHO	TTL for ICMP echo (RFC1700 says 64)
CFGITEM_IP_ICMPDONTREPLYBCAST	Do not Reply to ICMP Echo Request packets sent to broadcast/directed broadcast IP addresses (1 = Yes)
CFGITEM_IP_ICMPDONTREPLYMCAST	Do not Reply to ICMP Echo Request packets sent to multicast IP addresses (1 = Yes)
CFGITEM_IP_IPINDEXSTART	IP Protocol Start Index
CFGITEM_IP_IPFORWARDING	IP Forwarding Enable (1 = Yes)
CFGITEM_IP_IPNATENABLE	IP NAT Translation Enable (1 = Yes)
CFGITEM_IP_IPREASMMAXTIME	Maximum IP reassembly time in seconds
CFGITEM_IP_IPREASMMAXSIZE	Maximum IP reassembly packet size
CFGITEM_IP_DIRECTEDBCAST	Support directed BCast IP addresses (1 = Yes)
CFGITEM_IP_TCPREASMMAXPKT	Maximum out of order packets held by TCP socket
CFGITEM_IP_RTCENABLEDEBUG	Enable route control dbg messages (1 = Yes)
CFGITEM_IP_RTCADVTIME	Time in sec to send Router Adv. (0 = don't)
CFGITEM_IP_RTCADVLIFE	Lifetime of route in RtAdv if active
CFGITEM_IP_RTCADVPREF	Preference of route in RtAdv if active
CFGITEM_IP_RTARPDOWNTIME	Time 5 failed ARPs keeps route down
CFGITEM_IP_RTKEEPALIVETIME	Timeout of validated route in seconds
CFGITEM_IP_RTCLONETIMEOUT	Timeout of newly cloned route in seconds
CFGITEM_IP_RTDEFAULTMTU	MTU for internal routes
CFGITEM_IP_SOCKTTLDEFAULT	Default IP TTL for Sockets
CFGITEM_IP_SOCKTOSDEFAULT	Default IP TOS for Sockets
CFGITEM_IP_SOCKMAXCONNECT	Maximum connections on listening socket
CFGITEM_IP_SOCKTIMECONNECT	Maximum time for connect socket
CFGITEM_IP_SOCKTIMEIO	Default Maximum time for socket send/rcv
CFGITEM_IP_SOCKTCPTXBUF	TCP Transmit allocated buffer size
CFGITEM_IP_SOCKTCPRXBUF	TCP Receive allocated buffer size (copy mode)
CFGITEM_IP_SOCKTCPRXLIMIT	TCP Receive limit (non-copy mode)
CFGITEM_IP_SOCKUDPRXLIMIT	UDP/RAW Receive limit
CFGITEM_IP_SOCKMINTX	Default min Tx space for able to write

CFGITEM_IP_SOCKMINRX	Default min Rx data for able to read
CFGITEM_IP_PIPETIMEIO	Maximum time for pipe send/rcv call
CFGITEM_IP_PIPEBUFMAX	Pipe internal buffer size
CFGITEM_IP_PIPEMINTX	Pipe min Tx space for able to write
CFGITEM_IP_PIPEMINRX	Pipe min Rx data for able to read
CFGITEM_IP_TCPKEEPIDLE	Idle time before 1st TCP keep probe
CFGITEM_IP_TCPKEEPINTVL	TCP keep probe interval
CFGITEM_IP_TCPKEEPMAXIDLE	Maximum TCP keep probing time before drop
CFGITEM_IP_MAX	Maximum CFGTAG_STACK item

Network Tools Library - Support Functions

Included with the stack package is a library of network tools. It provides auxiliary functionality to the stack library and contains source written to the socket layer that would normally be considered application level code. The library file is called NETTOOLS.LIB, and can be accessed by an application that includes the file `NETTOOLS.H`.

The support supplied by NETTOOLS can be categorized into two classes: support functions and services. The support functions consist of a programming API that can aid the development of network applications, while services are servers that execute on the stack platform.

This section describes the NETTOOLS support functions.

Topic **Page**

5.1 Generic Support Calls

5.1.1 Synopsis

This section contains a selection of functions that can be very useful when programming network applications. Some are standard Berkeley Software Distribution (BSD) Socket APIs while others are custom to the stack - designed to save you the time and trouble of programming directly to the stack API.

5.1.2 Function Overview

The following is a summary of the support functions described in this section:

inet_addr()	Convert a string to a 32 bit IP address in network format
inet_aton()	Convert a string to an *in_addr* structure record
NtAddNetwork()	Add a host network to a logical interface handle
NtRemoveNetwork()	Remove a network added with *NtAddNetwork()*
NtAddStaticGateway()	Add a static gateway route to the route table
NtRemoveStaticGateway()	Remove a static gateway route
NtIfIdx2Ip()	Get the IP host address assigned to a physical interface Index
NtGetPublicHost()	Get the system public IP address and domain name
NtIPN2Str()	Convert 32 bit IP address in network format to string

5.1.3 Network Tools Support API Functions

inet_addr — ***Return 32-bit Binary Network Ordered IPv4 Address***

Syntax

IPN inet_addr(char *strptr);

Parameters

strptr	Pointer to character string

Return Value

IP address or NULL.

Description

This function converts an IP address printed in a character string to a 32-bit network ordered IP address value. Note that leading 0s in the address string are interpreted as octal. The function returns NULL on failure.

This function actually calls *inet_aton()*, which is the better form of the function.

inet_aton — *Convert IP Address from String and Return in in_addr Structure*

Syntax

int inet_aton(char *strptr, struct in_addr *pa);

Parameters

strptr	Pointer to character string
pa	Pointer to address structure

Return Value

1 on success or 0 on failure.

Description

This function converts an IP address printed in a character string to a 32-bit network ordered IP address value. Note that leading 0s in the address string are interpreted as octal. The function return writes the IP address into the *in_addr* structure pointed to by the *pa* parameter. The function returns 1 on success and 0 on failure.

NtAddNetwork — *Add Host Network to Interface by IF Handle*

Syntax

HANDLE NtAddNetwork(HANDLE hIF, IPN IPHost, IPN IPMask);

Parameters

hIF	Handle to target interface
IPHost	IP Host Address (in network format)
IPMask	IP Host Subnet Mask (in network format)

Return Value

Handle to network binding on success or NULL on failure.

Description

This function attempts to add the specified IP host address (and mask) to the specified logical interface handle. The function returns a handle to the binding that binds the IP address to the interface. On an error, the function returns NULL. The most common error would be that adding the host address caused a duplicate IP indication from another host.

Note: In place of this function, consider using the configuration system with the CFGTAG_IPNET configuration entry (see Section 4.4.4).

NtRemoveNetwork — *Remove Host Network from Interface*

Syntax

void NtRemoveNetwork(HANDLE hBind);

Parameters

hBind	Handle to network binding returned by *NtAddNetwork()*.

Return Value

None.

Description

This function removes a network that was previously added with *NtAddNetwork()*.

NtAddStaticGateway *Add Static Gateway Route to the Route Table*

Syntax

HANDLE NtAddStaticGateway(IPN IPDestAddr, IPN IPDestMask, IPN IPGateAddr);

Parameters

IPDestAddr	IP address of destination (in network format)
IPDestMask	IP subnet mask of destination (in network format)
IPGateAddr	IP address of next hop gateway (in network format)

Return Value

Handle to newly created route or NULL on error.

Description

This function adds a static gateway route to the system route table.

IPDestAddr is the IP base address of the IP network of the network that is made accessible via the IP gateway. This value should be pre-masked with the IPDestMask so that:

(IPDestAddr & IPDestMask) = IPDestMask

This is used as a sanity check by the system. For a default route, the value is zero.

IPDestMask is the mask of the IP network accessible by the IP gateway. For a host route, the value is 0xFFFFFFFF, while for a default route, the value is zero.

IPGateAddr is the IP address of the gateway through which the specified IP network is accessible. It must be an IP address that is available on a locally connected network, i.e., one gateway cannot point to another.

The function returns a handle to the route created by this configuration entry. All routes are represented as route handles internally to the stack. This is discussed further in the appendices at the end of this document. Note that the handle returned here is *not* referenced (see the appendix for more details). All it means for the purposes of this function is that the handle can be discarded by the caller. It will remain valid until the route is removed via *NtRemoveStaticGateway()*.

Note: In place of this function, consider using the configuration system with the CFGTAG_ROUTE configuration entry (see Section 4.4.5).

NtRemoveStaticGateway *Remove Static Gateway Route from the Route Table*

Syntax

int NtRemoveStaticGateway(IPN IPTarget);

Parameters

IPTarget	IP address of destination to remove (in network format)

Return Value

Returns 1 if the route was removed, or 0 if it was not found.

Description

This function removes a static gateway route from the system route table. It searches for the route by destination IP address and will remove the first matching static route it finds. Note that only routes with both the GATEWAY and STATIC flags set are considered for removal.

NtIfIdx2Ip — *Get the 32-bit Representation of the IP Address of an Interface Index*

Syntax

int NtIfIdx2Ip(uint IfIdx, IPN *pIPAddr);

Parameters

IfIdx	Index of physical interface
pIPAddr	Pointer to receive IP address

Return Value

Returns 1 if an address was found, or 0 if it was not found.

Description

This function obtains the first IP host address found that is assigned to the supplied interface Index. The host address (in network format) is written to the pointer *pIPAddr*.

NtGetPublicHost — *Get the System Public IP Address and Domain Name*

Syntax

int NtGetPublicHost(IPN *pIPAddr, uint MaxSize, UINT8 *pDomain);

Parameters

pIPAddr	Pointer to receive IP address
MaxSize	Size of string buffer pointed to by *pDomain*
pDomain	Pointer to string buffer to receive domain name

Return Value

Returns 1 if information was found, or 0 if it was not found.

Description

This function gets the best IP address and domain name to use for access to the external network. For determining the best address and domain name, public addresses and domain names are preferred over IP addresses and domain names of virtual networks. The IP address (in network format) is written to *pIPAddr*, and the domain name is copied to *pDomain*.

NtIPN2Str — *Convert 32-bit IP Address in Network Format to String*

Syntax

void NtIPN2Str(IPN IPAddr, char *pStrBuffer);

Parameters

IPAddr	IP address in network format
pStrBuffer	Pointer to receive IP address string

Return Value

None.

Description

This function performs a *sprintf()* of the IP address supplied in IPAddr to the buffer supplied in *pStrBuffer*. Note that no buffer size is provided. This is because the size is deterministic, and will not exceed 16 characters (including the NULL terminator).

5.2 DNS Support Calls

5.2.1 Synopsis

The concepts and code behind the Unix *gethostbyname()* and *gethostbyaddr()* functions is extensive, and there are public domain versions available, which can be easily run on the IP stack library.

Although the code to support the whole name, address and server database is quite large, the basic name resolution functions are quite useful. For this reason, the stack provides a basic form of these function calls, without incurring the overhead associated with a full implementation. The DNS resolver used by these client functions is the same as accessed by the DNS server. When the configuration contains client machine records (i.e., controls local domain names), these entries are checked when the matching domain is encountered. Otherwise (and for all other queries), the query is resolved via external DNS servers.

In addition to providing a more compact implementation, the calls provided here are reentrant, which is not true of the standard Unix counterparts.

5.2.2 Function Overview

The following is a summary of the support functions described in this section:

DNSGetHostname()	Return the hostname of the current host
DNSGetHostByAddr()	Resolve a hostname from an IP address
DNSGetHostByName()	Resolve a hostname and IP address from a hostname

5.2.3 Standard Types and Definitions

5.2.3.1 Host Entry Structure

The DNS client functions all take a pointer to a buffer. They treat this buffer as a pointer to a host entry structure. If the function takes a pointer to a scrap buffer, a host entry structure is allocated from the start of this scrap buffer. Thus, on successful return from one of these calls, the pointer to the scrap buffer may be treated as a pointer to a host entry structure.

The structure differs slightly from the conventional definition. It is defined as follows:

```
//
// Host Entry Structure
//
struct _hostent {
        char   *h_name;         // Official name of host
        int     h_addrtype;     // Address Type (AF_INET)
        int     h_length;       // Address Length (4)
        int     h_addrcnt;      // Number of IP addresses found
        IPN     h_addr[8];      // List of up to 8 IP addresses (network format)
        };

typedef struct _hostent HOSTENT;
```

5.2.3.2 Function Return Codes

DNS functions that return an error code use the following definitions. Those that are obtained directly from a DNS response packet are so noted:

NOERROR	0	(DNS Reply Code) No error
FORMERR	1	(DNS Reply Code) Format error
SERVFAIL	2	(DNS Reply Code) Server failure
NXDOMAIN	3	(DNS Reply Code) Non existent domain
NOTIMP	4	(DNS Reply Code) Not implemented
REFUSED	5	(DNS Reply Code) Query refused
OVERFLOW	16	Scrap Buffer Overflow
MEMERROR	17	Memory Allocation Error (used for packets and temp storage)
SOCKETERROR	18	Socket Error (call *fdError()* for socket error number)
NODNSREPLY	19	No DNS server response

5.2.4 DNS Support API Functions

DNSGetHostname — *Return the Hostname of the Current Host*

Syntax

int DNSGetHostname(char *pNameBuf, int size);

Parameters

pNameBuf	Pointer to a buffer to accept the hostname
size	Size of the supplied buffer in bytes

Return Value

Error code as defined above.

Description

This function is quite similar to BSD's *gethostname()*. It requests the hostname of the system's public IP address (as obtained from *NtGetPublicHost()*). The hostname is copied into the buffer pointed to by *pNameBuf* with a maximum size of *size*. The name is NULL terminated when space allows.

DNSGetHostByAddr *Resolve a Hostname from an IP Address*

Syntax

int DNSGetHostByAddr(IPN IPAddr, void *pScrapBuf, int size);

Parameters

IPAddr	IP address to resolve, in network format
pScrapBuf	Pointer to a scrap buffer from which a HOSTENT structure will be allocated
size	Size of the supplied scrap buffer in bytes

Return Value

Error code as defined above.

Description

This function is quite similar to BSD's *gethostbyaddr()*. It uses DNS to resolve a hostname from the supplied IP address. On a successful return, *pScrapBuf* can be treated as a HOSTENT structure. The size of the scrap buffer (*size*) must be greater than the size of the structure as the structure will contain pointers into the scrap buffer, and the scrap buffer is also used for temporary name storage. 512 bytes should be sufficient for most requests.

DNSGetHostByName *Resolve a Hostname/Address from a Hostname*

Syntax

int DNSGetHostByName(char *Name, void *pScrapBuf, int size);

Parameters

Name	Null terminated Hostname to resolve (with or without trailing '.')
pScrapBuf	Pointer to a scrap buffer from which a HOSTENT structure will be allocated
size	Size of the supplied scrap buffer in bytes

Return Value

Error code as defined above.

Description

This function is quite similar to BSD's *gethostbyname()*. It uses DNS to resolve an official hostname and address from the supplied hostname. On a successful return, pScrapBuf can be treated as a HOSTENT structure. The size of the scrap buffer (size) must be greater than the size of the structure as the structure will contain pointers into the scrap buffer, and the scrap buffer is also used for temporary name storage. 512 bytes should be sufficient for most requests.

If the hostname *Name* is terminated with a dot (.), the dot is removed prior to lookup. If a dot appears anywhere in *Name*, an initial lookup on the unaltered name is attempted. If *Name* does not contain a dot, or if the initial lookup fails, the default domain name (from *NtGetPublicHost()*) is appended to the end of the supplied name. For example, if the domain name obtained from *NtGetPublicHost()* was ti.com, then a request for host.sc would attempt to resolve host.sc first, and then host.sc.ti.com, while a request for host would attempt to resolve host.sc.ti.com on the initial attempt.

5.3 TFTP Support

5.3.1 Synopsis

TFTP is supported via the received function. More information on TFTP can be found in RFC783, released by the Internet Engineering Task Force (IETF) organization.

5.3.2 TFTP Support API Functions

TFTP is accessed through this API. The network tools include the file `NETTOOLS.H` , which is required.

NtTftpRecv ***Retrieve Data from a TFTP Server***

Syntax

int NtTftpRecv(UINT32 TftpIp, char *szFileName, char *pFileBuffer, UINT32 *pFileSize, UINT16 *pErrorCode);

Parameters

TftpIp	IP Address in network format
szFileName	Pointer to null terminated filename string
pFileBuffer	Pointer to buffer to receive file data
pFileSize	Pointer to size of buffer on input, returns as size needed or used
pErrorCode	Pointer to where to write TFTP server error code (if any)

Return Value

This function returns an error code indicating the results of the operation. Negative codes are error conditions.

In the following cases, *pFileSize* is set to the actual file size:

0	Successful transfer and copy
1	Successful transfer, with partial copy (file size too large)

In the following cases, *pFileSize* is set to the actual number of bytes copied:

TFTPERROR_ERRORREPLY	Error returned by TFTP server (see below)
TFTPERROR_BADPARAM	Invalid calling parameters
TFTPERROR_RESOURCES	Memory allocation error during transfer
TFTPERROR_SOCKET	Internal socket error during transfer
TFTPERROR_FAILED	TFTP failed (e.g., server did not reply)

In the case of TFTPERROR_ERRORREPLY, the server error code written to **pErrorCode* should be one of the following standard TFTP codes, and the error message is copied to **pFileBuffer*:

0	Not defined, see error message (if any).
1	File not found.
2	Access violation.
3	Disk full or allocation exceeded.
4	Illegal TFTP operation.
5	Unknown transfer ID.
6	File already exists.
7	No such user.

Description

TFTP (Trivial File Transfer Protocol), allows files to be transferred from a remote machine.

This function attempts to receive the file with the filename designated by *szFileName* from the TFTP server with the IP address in *TftpIp*, and copy the data into the memory buffer pointed to by *pFileBuffer*. Note that when specifying the name of the file in *szFileName*, certain operating systems have case sensitive naming conventions.

On entry, the parameter *pFileSize* must point to the size of the buffer pointed to by *pFileBuffer*. If the value at **pFileSize* is null, the *pFileBuffer* parameter can be NULL.

This function attempts to receive the entire file, even if the buffer space is insufficient. The return value indicates if the file was received.

A return value of 1 indicates that the file was received and copied into the buffer. A return value of 0 indicates that the file was received, but was too large for the specified buffer. In both these cases, the actual size of the file in bytes is written back to **pFileSize*.

A negative return value indicates that an error has occurred during transfer. In this case, the number of bytes actually consumed in the buffer is written back to **pFileSize*. An error return of TFTPERROR_ERRORREPLY is a special return value that indicates that an error code was returned from the TFTP server. In this case, the server's TFTP error code is written to **pErrorCode*, and the server's TFTP error message string is copied to the data buffer pointer to by *pFileBuffer*.

5.4 TCP/UDP Server Daemon Support

5.4.1 Synopsis

A server daemon is a single network task that monitors the socket status of multiple network servers. When activity is detected, the daemon creates a task thread specifically to handle the new activity. This is more efficient than having multiple servers, each with their own listening thread.

5.4.2 Server Daemon Support API Functions

Entries in the server daemon are created and destroyed through the following APIs. The network tools include the file `NETTOOLS.H` , which is required.

DaemonNew — *Create a New TCP/UDP Server Entry*

Syntax

HANDLE DaemonNew(uint Type, IPN LocalAddress, uint LocalPort, int (*pCb)(SOCKET,UINT32), uint Priority, uint StackSize, UINT32 Argument, uint MaxSpawn);

Parameters

Type	Socket type (SOCK_STREAM, SOCK_STREAMNC, or SOCK_DGRAM)
LocalAddress	Local IP address (set to NULL for wildcard)
LocalPort	Local Port to serve (cannot be NULL)
pCb	Pointer to callback to handle server event (connection or activity)
Priority	Priority of new task to create for callback function
StackSize	Stack size of new task to create for callback function
Argument	Argument (besides socket) to pass to callback function
MaxSpawn	Maximum number of callback function instances (must be 1 for UDP)

Return Value This function returns a handle to a daemon , or NULL on error.

Description Once a new entry is created, the daemon will create the desired TCP or UDP socket, and start listening for activity.

In the case of TCP, when a new connection is established, a new task thread is created, and a socket session is opened. Then the user's callback function is called on the new task thread, being supplied with both the socket to the new connection and the caller specified argument (as supplied to *DaemonNew()*). The callback function can keep the socket and task thread for as long as necessary. It returns from the callback once it is done with the connection. The function can choose to close the socket if desired. The return code informs the daemon whether the socket has been closed (0) or is still open (1).

In the case of UDP, when any data is available on the UDP socket, a new task thread is created, and a socket session is opened. Then the user's callback function is called on the new task thread, being supplied with both the UDP socket and the caller specified argument (as supplied to *DaemonNew()*). The callback function can keep the socket and task thread for as long as necessary. It returns from the callback only when it is done with the data. (While the callback function holds the UDP socket, the daemon will ignore further activity on it.) The callback should return 1, as it should not close the UDP socket.

DaemonFree — *Destroy a TCP/UDP Server Entry*

Syntax voidDaemonFree(HANDLEhEntry);

Parameters

hEntry	Handle to server entry returned from *DaemonNew()*

Return Value None.

Description Destroys a daemon entry, and closes the socket session of all child tasks spawned from the entry. Closing the socket sessions will result in all socket functions returning SOCKET_ERROR in all spawned child tasks. Thus, all spawned tasks should error out and return to the daemon, allowing them to be freed.

5.4.3 Server Daemon Example

The following is an example TCP echo server using the server daemon. The TCP server will use SOCK_STREAMNC for non-copy TCP. Its only job is to read from the socket, and write back what it reads.

To install the server on port 7, use the following code:

```
hEcho = DaemonNew( SOCK_STREAMNC, 0, 7, dtask_tcp_echo,
                   OS_TASKPRINORM, OS_TASKSTKNORM, 0, 3 );
```

This code allows up to three echo sessions to be running simultaneously on different threads. Note the IP specified is NULL, allowing echo connection on any local IP address assigned to the system.

To destroy the server and all its instances, the hEcho handle returned from *DaemonNew()* is used:

```
DaemonFree( hEcho );
```

The code for the callback function dtask_tcp_echo() is as follows:

```
int dtask_tcp_echo( SOCKET s, UINT32 unused )
{
    struct timeval to;
    int            I;
    char           *pBuf;
    HANDLE         hBuffer;

    (void)unused;
```

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

        I = 1;
        setsockopt( s, IPPROTO_TCP, TCP_NOPUSH, &I, 4 );


        for(;;)
        {
            I = (int)recvnc( s, (void **)&pBuf, 0, &hBuffer );

            // If we read data, echo it back
            if(I > 0)
            {
                if(send( s, pBuf, I, 0) < 0 )
                    break;
                recvncfree( hBuffer );
            }


            // If the connection got an error or disconnect, close
            else
                break;
        }
        fdClose( s );

        // Return "0" since we closed the socket
        return(0);
    }
```

Network Tools Library - Services

Included with the stack package is a library of network tools. It provides auxiliary functionality to the stack library and contains source written to the socket layer that would normally be considered application level code. The library file is called NETTOOLS.LIB, and can be accessed by an application that includes the file `NETTOOLS.H`.

The support supplied by NETTOOLS can be categorized into two classes: support functions and services. The support functions consist of a programming API that can help develop network applications, while services are servers that execute on the stack platform.

This section describes the NETTOOLS services.

Topic **Page**

6.1 Service Calling Conventions

6.1.1 Specifying Network Services Using the Configuration

Although each service has it own specific API, it is usually more convenient to add services by specifying the service in the system configuration as opposed to calling their individual Open and Close API functions. Included in the description of each network service is a description of its direct API, as well as an example of specifying the service in the system configuration.

6.1.1.1 Service Report Function

All the configuration examples in this section use a common service report callback function. The following is a very simple implementation of a service report function that calls printf() to print service status.

Note that this function relies on the physical value of items in the configuration specification found in the file: inc\nettools\netcfg.h.

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

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

6.1.2 Invoking Network Services by NETTOOLS API

Each service API uses a common calling format. This allows the services to be invoked by the configuration system using callback functions provided in the Network Control software (which also performs system initialization). It is preferable to launch services via the configuration system, instead of manually calling each Open and Close function described in the following sections. However, because the source to the Network Control software uses these calls, they are documented here.

The common calling interface consists of a simple Open and Close concept. The Open function initiates the service and returns a service handle, while the Close function shuts down the service using the service handle returned from the Open call.

Each service Open call takes at least one parameter. This parameter is a pointer to a common argument structure called NTARGS. The specification of this structure is as follows:

```
typedef struct _ntargs {
        int     CallMode;           // Determines desired calling mode
#define NT_MODE_IFIDX1              // Call by specifying IfIdx
#define NT_MODE_IPADDR2             // Call by specifying IPAddr
        int     IfIdx;              // Physical interface Index (0-n)
        IPN     IPAddr;             // IP Address
        HANDLE  hCallback;          // Handle to pass to callback function
        void(*pCb)(HANDLE, uint);   // Callback for status change
        } NTARGS;
```

Note that this entry structure is a simplified version of that provided by the configuration system. This structure also contains a callback function. The callback function is a subset of that in the configuration system, and codes returned by this callback are passed through the configuration callback to the application.

The individual fields are defined as follows:

- `int  CallMode;`

This parameter determines how the service is launched, either by IP address or by interface index (1 to n).

Some services can be launched either on a specific interface (1 to n) or on a specific IP address (which can also be the wildcard INADDR_ANY). Generally, any service that accepts an IP address can also accept an interface. The service will look up the IP address for the specified interface.

Other services can only be executed by interface and are independent of IP address. These are said to be compatible with NT_MODE_IFIDX only.

The value of *CallMode* can be one of the following:

NT_MODE_IFIDX	Call by specifying the interface Index (1 to n)
NT_MODE_IPADDR	Call by specifying IP address in network format

- `int  IfIdx;`

  This is the physical interface index (1 to n) on which the service is to be executed. For example, when launching a DHCP server service, the physical interface is that connected to the home network. For more generic services (like Telnet), the service can be launched by a pre-defined IP address (or INADDR_ANY as a wildcard). When launching by IP address only, this field is left NULL. When this field is used, *CallMode* should be set to NT_MODE_IFIDX.
- `IPN  IPAddr;`

  This is the IP address (in network format) on which to initiate the service. This IP address can specify the wildcard INADDR_ANY, in which case the service will accept connections to any valid IP address on any device. Note that some services (like DHCP server) do not support being launched by IP address. When this field is used, *CallMode* should be set to NT_MODE_IPADDR.
- `HANDLE hCallback;`

  This is the caller supplied handle that is passed back to the caller when the status callback function is invoked (see below).
- `void  (*pCb)(HANDLE, uint);`

  This is a pointer to a caller supplied callback function by which the service reports status.

  The specification of this callback is:

  ```
  void cbFun(HANDLE hCallback, uint NtStatus);
  ```

hCallback	Handle supplied to the service by the caller
NtStatus	NetTools Service Status code

The *NtStatus* parameter consists of an upper byte that is predefined, and a lower byte that is specific to the service. When masked with ~0xFF (NOT 0xFF), the value will be one of the following:

NETTOOLS_STAT_NONE.	Nothing reported
NETTOOLS_STAT_RUNNING	Service is initialized (running)
NETTOOLS_STAT_PARAMUPDATE	The service parameter structure has changed (the configuration containing this structure should be saved)
NETTOOLS_STAT_COMPLETED	The service has run to completion
NETTOOLS_STAT_FAULT	The service has halted due to a fault

Note that this callback function does not go directly to the application when using the configuration system. These codes are supplied to the configuration service callback in the *Code* parameter.

An optional second parameter to each service Open function is a pointer to a private service parameter structure. In the configuration section of this document, the individual service parameter structures were included in the specification of the configuration entry instance structure for each service.

6.2 Telnet Server Service

6.2.1 Synopsis

The Telnet Server service provides a mechanism for exposing a stream IO connection to any remote telnet client console.

A telnet connection is basically just a TCP connection to the well-known port designated for telnet. However, there is some data translation that occurs on the stream. Telnet has a set of commands that can change the behavior of the terminal, and can perform some character translation. The telnet server supplied here is designed to convert a normal TTY stream to a telnet stream and back. This allows any application to treat a telnet session as any other TTY session (like a serial port).

Connection to an application is achieved by use of an application supplied callback function that telnet calls when a new connection is established. This callback function returns the file descriptor of one end of a full duplex communications pipe. By allowing multiple calls to the callback function, console applications can be written to work with multiple IO streams.

6.2.2 Telnet Parameter Structure

The following structure defines the unique parameters of the Telnet service. It is located in the file: inc\nettools\inc\telnetif.h.

```
//
// Telnet Parameter Structure
//
typedef struct _ntparam_telnet {
        int     MaxCon;             // Max number of telnet connections
        int     Port;               // Port (set to NULL for telnet default)
        int     (*Callback)(PSA);   // Connect function returns local pipe
        } NTPARAM_TELNET;
```

MaxCon	Maximum number of simultaneous telnet sessions (1 to 24)
Port	TCP port to use for Telnet (set to zero for Telnet default)
Callback	Pointer to a callback function that takes a pointer to a sockaddr structure, and returns a local file descriptor to one end of a full duplex communications pipe

This structure is used both when specifying the service to the configuration system or when bypassing the configuration and invoking the service API directly.

6.2.3 Specifying Service Using the Configuration

The service can be specified as public because it can connect using any IP address, or an IP address of a specific interface. When accepting connections to any system IP address, the service is specified with the CALLBYIP flag and an IP address of INADDR_ANY. When a private connection is desired, the service is specified by the physical interface on which connections are allowed to occur. Because an IP address is required to initialize the service, the RESOLVEIP flag should also be set in the latter case.

For example, the following code specifies that the telnet server should run using the IP address INADDR_ANY.

```
telnet_example()
{
    CI_SERVICE_TELNET telnet;

    bzero( &telnet, sizeof(telnet) );
    telnet.cisargs.IPAddr = INADDR_ANY;
    telnet.cisargs.pCbSrv = &ServiceReport;
    telnet.param.MaxCon   = 2;
    telnet.param.Callback = &ConsoleOpen;
```

```
    CfgAddEntry( hCfg, CFGTAG_SERVICE, CFGITEM_SERVICE_TELNET, 0,
                 sizeof(telnet), (UINT8 *)&telnet, 0 );
}
```

The above code is all that is required when using the configuration system to invoke this service.

6.2.4 Invoking the Service via NETTOOLS API

In addition to the configuration option, this service can also be created and destroyed directly through this NETTOOLS API. If an application wishes to bypass the configuration system and launch the service directly, these calls can be used.

TelnetOpen — ***Create an Instance of the Telnet Server***

Syntax

HANDLE TelnetOpen(NTARGS *pNTA, NTPARM_TELNET *pNTP);

Parameters

pNTA	Pointer to common argument structure used by all services
pNTP	Pointer to Telnet parameter structure

Return Value

Returns a handle to the new telnet server instance, or NULL if the service could not be created. This handle is used with *TelnetClose()* to shut down the server when it is no longer needed.

Description

When a telnet session is established, a telnet child task is spawned that will call the supplied callback function. This callback function should return a local file descriptor of one end of a full duplex pipe. If the callback function returns -1, the connection is aborted. When either the terminal or telnet connection end of the pipe is broken, the other connection is closed and the session is ended.

TelnetClose — ***Destroy an Instance of the Telnet Server***

Syntax

void TelnetClose(HANDLE hTelnet);

Parameters

hTelnet	Handle to telnet server instance obtained from *TelnetOpen()*

Return Value

None.

Description

Destroys the instance of the telnet server indicated by the supplied handle. Once called, the server is shut down and no further telnet sessions can be established. Also, all spawned connections are immediately terminated.

6.3 DHCP Server Service

6.3.1 Synopsis

When acting as a router, the NDK may also need to maintain the network configuration on one of its network devices. A DHCP server allows the stack to maintain the IP address of multiple Ethernet client devices. When combined with Network Address Translation (NAT), the DHCP server can be used to establish client membership in a private virtual network.

6.3.2 Operation

The DHCP server can be optionally configured to allocate IP addresses out of a pool that is specified by an IP base address and the number of addresses in the pool. If no pool is specified, the server will use static client entries in the configuration system to resolve client address requests.

The server will respond to DHCP requests from a single Ethernet device. This allows for isolation of clients for a given interface, and allows multiple instances of the DHCP server to manage different IP address pools for different interfaces.

6.3.3 DHCP Server Parameter Structure

The following structure defines the unique parameters of the DHCP server service. It is located in the file: inc\nettools\inc\dhcpsif.h.

```
//
// DHCPS Parameter Structure
//
typedef struct _ntparam_dhcps {
        uint    Flags;          // DHCPS Execution Control Flags
        IPN     PoolBase;       // First IP address in optional pool
        uint    PoolCount;      // Number of addresses in optional pool
        } NTPARAM_DHCPS;
```

- Flags - Execution control flags. Can be any combination of the following:

DHCPS_FLG_LOCALDNS	Causes DHCPS to report its own IP address as the local DNS server to clients. If this flag is not set, DHCPS reports the DNS servers as contained in the SYSINFO portion of the configuration.
DHCPS_FLG_LOCALDOMAIN	Causes DHCPS to report the local domain name assigned to the virtual network to clients. If this flag is not set, DHCPS reports the public domain name to clients.

- PoolBase - The first IP address (in network format) of the address pool.
- PoolCount - The number of addresses in the address pool.

This structure is used both when specifying the service to the configuration system or when bypassing the configuration and invoking the service API directly.

6.3.4 Specifying Service Using the Configuration

Because the DHCP server service executes on a specific interface, it is never executed based on an IP address. Thus, it cannot be used with the CALLBYIP flag in the standard configuration service structure. However, because an IP host address is required to initialize the service on a specific interface, the RESOLVEIP flag should be set in cases where the IP address is not pre-assigned.

For example, the following code specifies that the DHCP server should run on the interface specified by the physical index *dhcpsIdx*. Here, the home networks have already been written to the configuration, so the RESOLVEIP flag is not necessary. The address pool being used is already stored in *IPPoolBase* and *PoolSize*. The DHCPS is requested to report the local server address as a DNS server to DHCP clients.

```
dhcp_server_example()
{
    CI_SERVICE_DHCPS dhcps;

    bzero( &dhcps, sizeof(dhcps) );
    dhcps.cisargs.Mode   = CIS_FLG_IFIDXVALID;
    dhcps.cisargs.IfIdx  = dhcpsIdx;
    dhcps.cisargs.pCbSrv = &ServiceReport;

    // Report our address as a DNS server to clients, and use the
    // network's local domain name.
    dhcps.param.Flags = DHCPS_FLG_LOCALDNS | DHCPS_FLG_LOCALDOMAIN;

    // Assign the IP address pool
    dhcps.param.PoolBase  = IPPoolBase;
    dhcps.param.PoolCount = PoolSize;
```

```
    CfgAddEntry( hCfg, CFGTAG_SERVICE, CFGITEM_SERVICE_DHCPSERVER, 0,
                 sizeof(dhcps), (UINT8 *)&dhcps, 0 );
}
```

The above code is all that is required when using the configuration system to invoke this service.

6.3.5 Invoking the Service via NETTOOLS API

In addition to the configuration option, this service can also be created and destroyed directly through this NETTOOLS API. If an application wishes to bypass the configuration system and launch the service directly, these calls can be used.

DHCPSOpen — *Open a DHCP Server*

Syntax

HANDLE DHCPSOpen(NTARGS *pNTA, NTPARAM_DHCPS *pNTP);

Parameters

pNTA	Pointer to common argument structure used by all services.
pNTP	Pointer to DHCP parameter structure

Return Value

Returns a HANDLE to a DHCPS instance structure that is used in calls to other DHCPS functions like *DHCPSClose()*.

Description

This function is called to initiate DHCPS control of an IP address pool on a given interface. The base address of the address pool does not have to be the first IP address in the subnet.

The DHCP Server executes on a specific interface. Thus, it is compatible with NT_MODE_IFIDX only.

DHCPSClose — *Close an Instance of the DHCP Server*

Syntax

void DHCPSClose(HANDLE hDHCPS);

Parameters

hDHCPS	Handle to a DHCP server instance obtained from *DHCPSOpen()*

Return Value

None.

Description

This function is called to terminate DHCPS control of the previously supplied interface. This call also destroys the supplied DHCP server instance handle *hDHCPS*.

6.4 DHCP Client Support

6.4.1 Synopsis

At system start up, the DHCP client will try and acquire an IP address from the DHCP servers available on the network.

Note that the client will accept the first IP address offered and that the INIT-REBOOT State (which requests a previously assigned IP address) is not currently implemented.

More information on DHCP can be found in RFC2131 and RFC2132, released by the Internet Engineering Task Force (IETF) organization.

6.4.2 Operation

The DHCP client is a special service that always executes immediately in a system. It is usually after the DHCP client obtains a public IP address that most of the other services in the system can initialize.

The DHCP client code makes more use of the service status report callback function than most of the other services. Recall from the beginning of this section that the least significant byte of the report code is reserved for service specific information.

The following report codes are returned in the LSB of the report code sent by the DHCP service:

DHCPCODE_IPADD	An IP client address had been added to the system
DHCPCODE_IPREMOVE	An IP client address has been removed from the system
DHCPCODE_IPRENEW	An IP client address has been renewed

Note that in each of the above cases, the DHCP portion of the system information configuration (the first 256 entries of CFGTAG_SYSINFO) has been erased and potentially reprogrammed. If an application needs to share the DHCP portion of the system information configuration, these DHCP report codes can be used to signal when to add additional application specific tags. For more information on DHCP and the CFGTAG_SYSINFO tag, see Section 4.4.8.

6.4.3 DHCP Client Parameter Structure

The following structure defines the unique parameters of the DHCP client service. It is located in the file: inc\nettools\inc\dhcpif.h.

```
//
// DCHP Parameter Structure
//
#define DHCP_MAX_OPTIONS        64  // Max number of allowed options

typedef struct _ntparam_dhcp {
    UINT8   *pOptions;              // Options to request
    int     len;                    // Length of options list
    } NTPARAM_DHCP;
```

pOptions	Pointer to additional DHCP option tags to request. The list is used when additional information must be obtained from the DHCP server. Up to DHCP_MAX_OPTIONS tags can be specified. This pointer can be NULL when *len* is set to 0.
len	Specifies the length in bytes of the list pointed to by pOptions.

This structure is used both when specifying the service to the configuration system or when bypassing the configuration and invoking the service API directly.

6.4.4 Specifying Service Using the Configuration

Because the DHCP client service executes on a specific interface, it is never executed based on an IP address. Thus, it cannot be used with the CALLBYIP flag in the standard configuration service structure. Also, because the service runs without an IP host address, the RESOLVEIP flag should never be set.

For example, the following code specifies that the DHCP client should run on the interface specified by the physical Index *dhcpIdx*.

```
dhcp_client_example()
{
    CI_SERVICE_DHCPC dhcpc;

    bzero( &dhcpc, sizeof(dhcpc) );
    dhcpc.cisargs.Mode   = CIS_FLG_IFIDXVALID;
    dhcpc.cisargs.IfIdx  = dhcpIdx;
    dhcpc.cisargs.pCbSrv = &ServiceReport;

    CfgAddEntry( hCfg, CFGTAG_SERVICE, CFGITEM_SERVICE_DHCPCLIENT, 0,
                 sizeof(dhcpc), (UINT8 *)&dhcpc, 0 );
}
```

The above code is all that is required when using the configuration system to invoke this service.

6.4.5 Invoking the Service via NETTOOLS API

In addition to the configuration option, this service can also be created and destroyed directly through this NETTOOLS API. If an application wishes to bypass the configuration system and launch the service directly, these calls can be used.

DHCPOpen — *Open a DHCP Server*

Syntax

HANDLE DHCPOpen(NTARGS *pNTA , NTPARAM_DHCP *pNTP);

Parameters

pNTA	Pointer to common argument structure used by all services
pNTP	Pointer to DHCP parameter structure

Return Value

Returns a HANDLE to a DHCP instance structure, which is used in calls to other DHCP functions like *DHCPClose()*.

Description

This function is called to initiate DHCP control of a given device.

DHCPOpen() starts the DHCP process. This process will discover if there are any DHCP servers on the network and request an IP address. The result of the search for an IP address will be passed to the application via the standard network tools status callback.

The Client will remain running so it can renew the IP address when necessary.

For any additional option tags entered into the DHCP client parameter structure, the resulting information from the DHCP server is written to the system configuration under the CFGTAG_SYSINFO entry. See Section 4.4.8 for more information.

The DHCP Client executes on a specific interface. Thus, it is compatible with NT_MODE_IFIDX only.

DHCPClose — *Close an Instance of the DHCP Client*

Syntax

void DHCPClose(HANDLE hDHCP);

Parameters

	hDHCP	Handle to a DHCP server instance obtained from *DHCPSOpen()*

Return Value None.

Description This function is called to terminate DHCP control of the previously supplied interface and frees the supplied DHCP server instance handle *hDHCP*.

Note this function will also remove any IP address it has added to the system. In the case of a service shutdown, there will be no status callback indicating the address removal.

6.5 HTTP Server Support

6.5.1 Synopsis

An HTTP (Hypertext Transfer Protocol) Server allows a remote browser to view content on the server file system. Files can be stored for viewing and forms can also be stored to allow remote interaction with the system. Form POST functions become calls to application defined C functions that allow the embedded system to be remotely controlled via a HTTP browser.

6.5.2 Operation

The HTTP Server service provides a mechanism for serving HTTP content to remote HTTP client applications. It uses the Embedded File System contained in the OS adaptation layer. These functions in the EFS programming API include a prefix of efs_. Modifying the EFS functions in the OS adaptation layer allows the system programmer to support a variety of file storage options, including memory, flash cards and hard drives.

6.5.3 HTTP Server Parameter Structure

The following structure defines the unique parameters of the HTTP server service. It is located in the file: inc\nettools\inc\httpif.h.

```
//
// HTTP Parameter Structure
//
typedef struct _ntparam_http {
        int     MaxCon;             // Max number of HTTP connections
        int     Port;               // Port (set to NULL for HTTP default)
        } NTPARAM_HTTP;
```

MaxCon	Maximum number of simultaneous telnet sessions (1 to 24)
Port	TCP port to use for HTTP (set to zero for HTTP default)

This structure is used both when specifying the service to the configuration system or when bypassing the configuration and invoking the service API directly.

6.5.4 Using the HTTP Server and Adding Web Content

This section discusses how to invoke and monitor the status of the HTTP server. Web application developers will be more interested in how to add Web content, including HTML pages and CGI functions. These topics are discussed in Chapter E.

6.5.5 Specifying Service Using the Configuration

The service can be specified as public as it can connect using any IP address, or an IP address of a specific interface. When accepting connections to any system IP address, the service is specified with the CALLBYIP flag and an IP address of INADDR_ANY. When a private connection is desired, the service is specified by the physical interface on which connections are allowed to occur. Because an IP address is required to initialize the service, the RESOLVEIP flag should also be set in the latter case.

For example, the following code specifies that the HTTP server should run using the IP address INADDR_ANY.

```
http_example()
{
    CI_SERVICE_HTTP http;

    bzero( &http, sizeof(http) );
    http.cisargs.IPAddr = INADDR_ANY;
    http.cisargs.pCbSrv = &ServiceReport;

    CfgAddEntry( hCfg, CFGTAG_SERVICE, CFGITEM_SERVICE_HTTP, 0,
                 sizeof(http), (UINT8 *)&http, 0 );
}
```

The above code is all that is required when using the configuration system to invoke this service.

6.5.6 Invoking the Service via NETTOOLS API

In addition to the configuration option, this service can also be created and destroyed directly through this NETTOOLS API. If an application wishes to bypass the configuration system and launch the service directly, these calls can be used.

httpOpen — *Start the HTTP Server*

Syntax

HANDLE httpOpen(NTARGS *pNTA, NTPARAM_HTTP *pNTP);

Parameters

pNTA	Pointer to common argument structure used by all services.
pNTP	Pointer to HTTP client parameter structure.

Return Value

Returns a handle to the HTTP Server instance, or NULL if the HTTP Server task could not be created. This handle is used with *httpClose()* to shut down the client when it is no longer needed.

Description

httpOpen() starts the HTTP server process. This process will create a connection to the HTTP Port and listen. When a connection is made, another task will be created to service the request.

httpClose — *Destroy an instance of the HTTP Server*

Syntax

void httpClose(HANDLE hHTTP);

Parameters

hHTTP	Handle to a HTTP server instance obtained from *httpOpen()*

Return Value

None.

Description

Destroys the instance of the HTTP Server indicated by the supplied handle. Once called, the Server is shut down.

6.6 DNS Server Service

6.6.1 Synopsis

The DNS server service allows clients on a home network to resolve host names and addresses for clients on both the home and public networks.

6.6.2 Operation

The NDK contains a small DNS resolver that can resolve hostnames and addresses that are local to the system via the configuration, or those outside the system by using an external DNS server.

The DNS server service described here allows the same internal DNS resolver to be accessed by clients on a virtual (home) network. This allows clients on a home network to look up peers on the home network using the same DNS server that is used for external lookups. Thus, DNS service for the home network is transparent to these clients.

Because the DNS server service uses the same internal DNS resolver as the client services discussed earlier, the server adds very little overhead to the system.

6.6.3 DNS Server Parameter Structure

The DNS server service does not require a parameter structure.

6.6.4 Specifying Service Using the Configuration

The service can be specified as public because it can connect using any IP address, or an IP address of a specific interface. When accepting connections to any system IP address, the service is specified with the CALLBYIP flag and an IP address of INADDR_ANY. When a private connection is desired, the service is specified by the physical interface on which connections are allowed to occur. Because an IP address is required to initialize the service, the RESOLVEIP flag should also be set in the latter case.

For example, the following code specifies that the server should run using the IP address INADDR_ANY.

```
dns_server_example()
{
    CI_SERVICE_DNSSERVER dnss;

    bzero( &dnss, sizeof(dnss) );
    dnss.cisargs.IPAddr = INADDR_ANY;
    dnss.cisargs.pCbSrv = &ServiceReport;

    CfgAddEntry( hCfg, CFGTAG_SERVICE, CFGITEM_SERVICE_DNSSERVER, 0,
                 sizeof(dnss), (UINT8 *)&dnss, 0 );
}
```

The above code is all that is required when using the configuration system to invoke this service.

6.6.5 Invoking the Service via NETTOOLS API

In addition to the configuration option, this service can also be created and destroyed directly through this NETTOOLS API. If an application wishes to bypass the configuration system and launch the service directly, these calls can be used.

DNSServerOpen *Create an Instance of the DNS Server*

Syntax

HANDLE DNSServerOpen(NTARGS *pNTA);

Parameters

pNTA	Pointer to common argument structure used by all services.

Return Value

Returns a handle to the new server instance, or NULL if the service could not be created. This handle is used with *DNSServerClose()* to shut down the server when it is no longer needed.

Description

Creates a DNS server task that can service external DNS requests using UDP.

DNSServerClose *Destroy an Instance of the DNS Server*

Syntax

void DNSServerClose(HANDLE hDNSS);

Parameters

hDNSS	Handle to DNS server instance obtained from *DNSServerOpen()*

Return Value

None.

Description

Destroys the instance of the DNS server indicated by the supplied handle. Once called, the server is shut down. It waits for all spawned sessions to complete.

6.7 Network Address Translation (NAT) Service

6.7.1 Synopsis

The NAT service allows for the establishment of a home virtual network that is isolated and protected from the external public network. It provides a port based address translation function that allows all the clients on the home network to share a single public IP address. Thus, multiple clients can share the same ISP account.

6.7.2 Operation

The NDK contains both a network address translation module and an IP filtering model. When the translation service is enabled, any packet received from a client on a virtual network that is destined for the external public network is adjusted to use the stack's public IP client address.

The translation is performed by allocating a translation record and holding it for a period of time. The translation records are timed out based on their protocol. In TCP, records are timed out based on the state of their TCP connection. UDP and ICMP translations time out based on when they were last used.

In addition to translation, the stack contains an IP filter option (always enabled by this service) that filters packets from the public network from being seen by the private network. For example, if someone on a public network knew the IP address and the subnet mask of the router's (stack in route mode) private network, it could set a gateway route to the router's public IP host address and the router would route packets from the public to the private network and back (internally it does not distinguish between public and private while routing). The IP filter prevents this. It also prevents an entity on a public network from accessing protocol servers (like HTTP or Telnet) that are running on the private network. This allows the router to present different HTTP or Telnet interfaces to the public than it does to clients in the home.

The NAT service is executed on the public interface - i.e., the interface that is assigned a valid public IP host address (used to carry traffic for the virtual client addresses). There can only be one instance and thus only one public IP address, but the service can serve multiple virtual (home) networks in the system so long as they can be combined and still exclude the public IP. If the combination of these networks results in an overlap with the public network, the service fails.

For example, assume interface If-1 is connected to the physical network 128.32.12.x/255.255.255.0, and there are two home networks (192.168.0.x/255.255.255.0) on If-2 and (192.168.1 .x/255.255.255.0) on If-3. To run NAT on both home networks, the NAT interface would be If-1 (the public interface), and the NAT group (virtual) network would be 192.168.0.0/255.255.254.0, which covers both home networks.

For more information on NAT operation, including how to program proxy filters, see Chapter B, *Network Address Translation*.

6.7.3 NAT Server Parameter Structure

The following structure defines the unique parameters of the NAT server service. It is located in the file: inc\nettools\inc\natif.h.

```
//
// NAT Parameter Structure
//
typedef struct _ntparam_nat {
        IPN     IPVirt;         // Virtual IP address
        IPN     IPMask;         // Mask of virtual subnet
        uint    MTU;            // NAT packet MTU (normally 1500 or 1492)
        } NTPARAM_NAT;
```

IPVirt	NAT Group virtual network address
IPMask	Subnet mask of NAT Group virtual network
MTU	IP MTU Limit (1500 for Ethernet, 1492 for PPPoE, etc.)

This structure is used both when specifying the service to the configuration system or when bypassing the configuration and invoking the service API directly.

6.7.4 Specifying Service Using the Configuration

Because the NAT service executes on a specified *public* interface, it is never executed based on an IP address. Thus, it cannot be used with the CALLBYIP flag in the standard configuration service structure. In addition, because the public IP host address is required to initialize the service, the RESOLVEIP flag should be set when the IP address is not pre-assigned.

For example, the following code specifies that the NAT service should run on the interface specified by the physical index *natIdx*. Here, the DHCP client service is used to obtain the public IP address (the address assigned to *natIdx*), so at this point the IP address is unknown. Thus, the RESOLVEIP flag is set in the execution mode parameter. This informs the configuration service manager not to invoke NAT until it has resolved an IP address for the target interface. The RESTART flag is also set to tell the service to restart NAT if a public IP address is lost and regained. In this example, it is assumed that all networks in the 192.168.x.x/255.255.0.0 subnet are part of the NAT group to be translated.

The MTU parameter to the NAT configuration allows the programmer to set a limit on the MTU negotiated during a TCP connection. This prevents TCP packet traffic from being unnecessarily fragmented. For example, when routing between Ethernet and PPPoE over NAT, the MTU should be set to the smaller MTU of the two, which is PPPoE's limit of 1492. In the example below, it is assumed that the system is Ethernet to Ethernet, and thus, it uses the full 1500.

```
nat_service_example()
{
    CI_SERVICE_NAT nat;

    bzero( &nat, sizeof(nat) );

    // Do not start NAT until we resolve an IP address on its IF
    nat.cisargs.Mode   = CIS_FLG_IFIDXVALID |
                         CIS_FLG_RESOLVEIP | CIS_FLG_RESTARTIPTERM;
    nat.cisargs.IfIdx  = natIdx;
    nat.cisargs.pCbSrv = &ServiceReport;
```

```
    // Include all 192.168.x.x addresses in NAT group
    nat.param.IPVirt = htonl(0xc0a80000);
    nat.param.IPMask = htonl(0xffff0000);
    nat.param.MTU    = 1500;

    CfgAddEntry( hCfg, CFGTAG_SERVICE, CFGITEM_SERVICE_NAT, 0,
                 sizeof(nat), (UINT8 *)&nat, 0 );
}
```

The above code is all that is required when using the configuration system to invoke this service.

6.7.5 Invoking the Service via NETTOOLS API

In addition to the configuration option, this service can also be created and destroyed directly through this NETTOOLS API. If an application wishes to bypass the configuration system and launch the service directly, these calls can be used.

NATOpen — ***Enable the NAT Service***

Syntax

HANDLE NATOpen(NTARGS *pNTA, NTPARAM_NAT *pNTP);

Parameters

pNTA	Pointer to common argument structure used by all services.
pNTP	Pointer to NAT parameter structure.

Return Value

Returns a handle to the NAT instance (1), or NULL if the service could not be created. This handle is used with *NATClose()* to disable the service when it is no longer needed.

Description

Enables the Network Address Translation Service. Although the function returns a handle for compatibility with the standard NETTOOLS API, only one instance of the NAT service is allowed.

This service utilizes the virtual and external network information using the configuration system. If the configuration system was not used to create the network records, this function will fail.

The NAT service executes on a specific public interface. Thus, it is compatible with NT_MODE_IFIDX only.

NATClose — ***Disable the NAT Service***

Syntax

void NATClose(HANDLE hNAT);

Parameters

hNAT	Handle to NAT service obtained from *NATOpen()*

Return Value

None.

Description

Disables the NAT service.

In the source code to the network control functions, there are several calls to internal stack functions. This is similar to calling the kernel in other operating environments. This section contains a partial list of internal stack functions provided to aid in the comprehension of kernel oriented calls.

Note the following points for this section:

1. This section is required only for system programming that needs low level access to the stack for configuration and monitoring. **This API does not apply to general sockets application programming.**
2. In addition to the internal functions described here, there are scheduling and configurations tools available that make any direct coding to these functions unnecessary.

Topic **Page**

A.1 Overview

The control API is the collection of functions supplied in the stack library. The entire API is exposed, although the vast majority of functions and objects will only be used internally to the stack.

A.1.1 Interrupts and Preemption

It should be noted that no part of the stack is interrupt driven. Neither can any stack function be called at interrupt time. All interrupt processing is performed in the HAL or OS libraries, and is thus externally-defined code, which allows the development of a HAL/OS architecture that is best suited for a given operating environment, without affecting the operation of the stack.

The stack may or may not be preempted, depending on the operating environment in use. A non-preemptive architecture is possible because the stack code does not use polling loops nor make any internal blocking type calls, but preemption is also supported.

A.1.2 Proper Use of the llEnter() and llExit() Functions

The internal stack functions are not designed to be reentrant. This allows the stack to operate freely without the concept of a critical section, which is implementation dependent and potentially detrimental to real-time operation. Thus, access to stack functions must be strictly controlled. The form of this control is dependant on the system environment, and is embodied as two low level OS library functions, *llEnter()* and *llExit()*. These functions are called before and after a section of code where any stack functions are called. For example:

```
llEnter();
StackFunction1();
StackFunction2();
llExit();
```

These functions can be thought of as entering and exiting kernel mode.

To make normal user functions appear to be re-entrant, some user functions (like the sockets API) make internal calls to *llEnter()* and *llExit()* when calling into the stack. If an application needs to call both user functions and internal stack functions, care must be taken so that standard user functions are not called between an *llEnter()* / *llExit()* pair (this would cause an error if they in turn called *llEnter()*).

The following are good general guidelines:

- Always call *llEnter()* before calling a stack function, and *llExit()* when done calling stack functions.
- Try and keep all code that requires *llEnter()* and *llExit()* in a single module. They are only required for system maintenance.
- Do not call a normal user function (like a socket function) between an *llEnter()*/*llExit()* pair.
- Never call *llEnter()* or *llExit()* from an ISR.

A.1.3 Objects

Many of the control API functions deal with object handles. These handles are created by a variety of class functions contained in the stack. When using an object handle, it is important to realize how the object handle will be treated by the function being called.

Associated with every object is the concept of who owns it, who is using it, and who will eventually free it. In general, when an application creates an object, the application owns it, the application is the only one using it, and the application must eventually free it. Unfortunately, the matter becomes somewhat confused when object handles are shared between applications — especially when the scope of the handle creator may be shorter than the handle itself.

In this system, there are two basic object types:

- Static Objects - The static object is one that is created by a designated task, and destroyed by that task or a task where the object has been passed. In most cases, the task that created the object also destroys it.
- Referenced Objects - A referenced object is one that may be used by other tasks after the original creator is through with it. This type of handle is useful when an object is needed for a task of indeterminate length, where the creator of the handle does not need or may not be able to track it.

  Under the referenced handle scheme, all tasks that access the object handle make a specific Ref*Xxx*() call so that references may be tracked. Whenever a task is finished with the handle, it calls the object's de-reference function. The object is not freed until the reference count reaches zero.

A.2 Stack Executive (Exec)

A.2.1 Synopsis

At the heart of the stack is the Executive API (Exec). The Executive acts as a message dispatcher for the internal stack components. This action is mostly hidden from the application, but there are some public functions.

A.2.2 API Functions

ExecOpen — ***Prepare the System for Execution***

Syntax void ExecOpen();

Description Prepares the stack for execution by initializing the individual components. Until *ExecOpen()* is called, the system cannot do any work, but after calling this function, objects like routes and bindings can be created.

ExecClose — ***Shutdown Stack and Cleanup***

Syntax void ExecClose();

Description Completes stack execution. This function is called to perform final clean up on the system after all user objects (like devices and bindings) have been destroyed.

ExecLowResource — ***Signal Low Resource Condition***

Syntax void ExecLowResource();

Description Informs the stack that memory resources are getting dangerously low. As a result of this call, the stack will abandon certain operations that hold excessive resources. (Pending ARP packets are thrown away, IP packet fragments pending reassembly are abandoned, etc.)

ExecTimer — ***Signal 1/10th Second Timer Tick***

Syntax void ExecTimer();

Description This function is called ten times a second to inform the stack that one tenth of a second has elapsed. This function is called from a normal task thread, never an ISR. In theory, the function can be called from anywhere, but in practice, it is always called from a scheduler thread that also handles network packets. For more information, see the description of the NETCTRL functions in the *TMS320C6000 Network Developer's Kit (NDK) Software User's Guide* (SPRU523).

A.3 Packet Buffer Manager (PBM) Object

A.3.1 Synopsis

The NDK uses a common packet buffer object that is managed by a module called the packet buffer manager (PBM). The implementation of this manager determines the buffer strategy for the entire system.

Internally, the packet buffer objects are pointers to a structure of type PBM_Pkt; however, the buffers are abstracted into a handle of type PBM_Handle for use by code outside of the NDK. This helps protect the reserved members of the packet buffer structure from being misused.

A.3.2 Object Type

Static - PBM objects are owned by a single entity and destroyed by their owner. Ownership of a packet buffer changes as it is passed via function calls.

A.3.3 API Function Overview

The PBM API functions are as follows:

Initialization/Shutdown Functions:

PBM_open()	Open the Packet Buffer Manager
PBM_close()	Close the Packet Buffer Manager

Create/Destroy Functions:

PBM_alloc()	Create New Packet Buffer
PBM_free()	Destroy (Free) Packet Buffer
PBM_copy()	Create an exact copy of the Packet Buffer

Property Functions:

PBM_getBufferLen()	Get the length of the physical data buffer
PBM_getDataBuffer()	Get a pointer to the physical data buffer
PBM_getValidLen()	Get the length of the valid data in the buffer
PBM_getDataOffset()	Get the buffer offset to the start of the valid data
PBM_getIFRx()	Get the device handle of the ingress Ethernet device
PBM_setValidLen()	Set the length of the valid data in the buffer
PBM_setDataOffset()	Set the buffer offset to the start of the valid data
PBM_setIFRx()	Set the device handle of the ingress Ethernet device

A.3.4 API Function Description

PBM_open — *Open the Packet Buffer Manager*

Syntax

uint PBM_open();

Parameters

None.

Return Value

Function returns 1 on success, and 0 on failure.

Description

This function is called once to open the PBM module and allow it to initialize its internal queues.

PBM_close — *Close the Packet Buffer Manager*

Syntax

void PBM_close();

Parameters

None.

Return Value

None.

Description

This function is called at system shutdown to allow the PBM module to shut down and free any memory it has allocated.

PBM_alloc — *Create New Packet Buffer*

Syntax

PBM_Handle PBM_alloc(uint MaxSize);

Parameters

MaxSize	Maximum size of the physical data buffer required

Return Value

Handle to the packet buffer or NULL on memory allocation error.

Description

This function is called to create a new packet buffer handle. When first created, the packet is entirely uninitialized, except for the physical characteristics of the data buffer (the buffer pointer and its physical length). The length of the buffer will be the same or greater than that specified by the caller in *MaxSize*.

PBM_free — *Destroy (Free) Packet Buffer*

Syntax

void PBM_free(PBM_Handle hPkt);

Parameters

hPkt	Handle to packet buffer to free

Return Value

None.

Description

This function is called to destroy a packet buffer. When called, all objects associated with the packet buffer are dereferenced or destroyed.

PBM_copy — *Create an exact copy of the Packet Buffer*

Syntax

PBM_Handle PBM_copy(PBM_Handle hPkt);

Parameters

hPkt	Handle to packet buffer to copy

Return Value

Handle to the new copy of the packet buffer or NULL on memory allocation error.

Description

This function makes a duplicate copy of a packet buffer. It is usually called to copy a packet to be distributed to multiple destinations, or to be sent to multiple egress devices.

PBM_getBufferLen — *Get the Length of the Physical Data Buffer*

Syntax

uint PBM_getBufferLen(PBM_Handle hPkt);

Parameters

hPkt	Handle to packet buffer

Return Value

Length of the physical data buffer in bytes.

Description

This function is called to get the length of the physical data buffer associated with the packet buffer handle. Note that the buffer length is fixed for the life of the buffer and cannot be changed.

PBM_getDataBuffer — *Get a Pointer to the Physical Data Buffer*

Syntax

UINT8 * PBM_getDataBuffer(PBM_Handle hPkt);

Parameters

hPkt	Handle to packet buffer

Return Value

Pointer to the physical data buffer.

Description

This function is called to get a pointer to the physical data buffer associated with the packet buffer handle. Note that the physical buffer is fixed and cannot be changed.

PBM_getValidLen — *Get the Length of the Valid Data in the Buffer*

Syntax

uint PBM_getValidLen(PBM_Handle hPkt);

Parameters

hPkt	Handle to packet buffer

Return Value

Byte length of the valid data stored in the packet buffer.

Description

This function is called to get the length of the valid data currently held in the packet buffer. When a packet buffer is created, it has no valid data, so this value is initially zero.

PBM_getDataOffset *Get the Buffer Offset to the start of the Valid Data*

Syntax

uint PBM_getDataOffset(PBM_Handle hPkt);

Parameters

hPkt	Handle to packet buffer

Return Value

Byte offset from the start of the physical data buffer to the first byte of valid data.

Description

This function is called to get the offset in bytes from the start of the physical data buffer to the first byte of valid data. When a packet buffer is created, it has no valid data, so this value is initially zero.

PBM_getIFRx *Get the Device Handle of the Ingress Ethernet Device*

Syntax

HANDLE PBM_getIFRx(PBM_Handle hPkt);

Parameters

hPkt	Handle to packet buffer

Return Value

NULL for locally created packets, or a handle to the device on which the packet was received.

Description

This function is called to get the handle to the ingress device where the packet contained in the packet buffer originated. Packet drivers in the HAL (both serial and Ethernet based) record the logical handle associated with all incoming packets. This identifies the packet type as well as the interface on which the packet was received.

PBM_setValidLen *Set the Length of the Valid Data in the Buffer*

Syntax

void PBM_setValidLen(PBM_Handle hPkt, uint length);

Parameters

hPkt	Handle to packet buffer
length	Length of the valid data held in the packet buffer

Return Value

None.

Description

This function is called to set the length of the valid data in the packet buffer. It informs the system of the number of bytes of valid data that are stored in the physical data buffer. When a packet buffer is created, it has no valid data, so this value is initially zero.

PBM_setDataOffset *Set the Buffer Offset to the Start of the Valid Data*

Syntax

void PBM_setDataOffset(PBM_Handle hPkt, uint offset);

Parameters

hPkt	Handle to packet buffer
offset	Offset from start of data buffer to valid data

Return Value

None.

Description

This function is called to set the offset in bytes from the start of the physical data buffer to the first byte of valid data. It informs the system of where valid data is stored in the physical data buffer. When a packet buffer is created, it has no valid data, so this value is initially zero.

PBM_setIFRx *Set the Device Handle of the Ingress Ethernet Device*

Syntax

void PBM_getIFRx(PBM_Handle hPkt, HANDLE hDevice);

Parameters

hPkt	Handle to packet buffer
hDevice	Handle to packet ingress device

Return Value

None.

Description

This function is called to set the handle to the ingress device where the packet contained in the packet buffer originated. Packet drivers in the HAL (both serial and Ethernet based) record the logical handle associated with all incoming packets. This identifies the packet type, as well as the interface on which the packet was received.

A.4 Packet Buffer Manager Queue (PBMQ) Object

A.4.1 Synopsis

The PBM module also includes a queue object that can be used to queue packet buffers for later use. The queue is a first in first out system, so it can be used to queue in-order packets as well as free buffers.

The PBMQ object is just a structure of type PBMQ. Once this structure is declared and initialized, it is ready for use.

A.4.2 Object Type

Static - PBMQ objects are owned by a single entity and destroyed by their creator.

A.4.3 API Function Overview

The PBM API functions are as follows:

PBMQ_init()	Initialize a PBMQ object for use
PBMQ_count()	Return the number of PBM packet buffers on the queue
PBMQ_enq()	Enqueue a PBM packet buffer onto the queue
PBMQ_deq()	Dequeue a PBM packet buffer off the queue

A.4.4 API Function Description

PBMQ_init — ***Initialize a PBMQ Object for Use***

Syntax

void PBM_init(PBMQ *pQ);

Parameters

pQ	Pointer to a structure of type PBMQ

Return Value

None.

Description

This function is called once to initialize a PBMQ structure for use.

PBMQ_count — ***Return the Number of PBM Packet Buffers on the Queue***

Syntax

uint PBM_count(PBMQ *pQ);

Parameters

pQ	Pointer to a structure of type PBMQ

Return Value

Number of queued buffers.

Description

This function is called once to return the number of PBM packet buffers currently on the indicated queue.

PBMQ_enq — ***Enqueue a PBM Packet Buffer onto the Queue***

Syntax

void PBM_enq(PBMQ *pQ, PBM_Handle hPkt);

Parameters

pQ	Pointer to a structure of type PBMQ
hPkt	Handle to PBM packet buffer to add to queue

Return Value

None.

Description

This function is called to add the supplied PBM packet buffer to the indicated queue.

PBMQ_deq — ***Dequeue a PBM Packet Buffer Off the Queue***

Syntax

PBM_Handle PBM_deq(PBMQ *pQ);

Parameters

pQ	Pointer to a structure of type PBMQ

Return Value

Handle to PBM packet buffer, or NULL on empty queue.

Description

This function is called to remove a PBM packet buffer from the indicated queue. The function returns a handle to the PBM packet buffer removed from the queue, or NULL if the queue was empty.

A.5 Stack Event (STKEVENT) Object

A.5.1 Synopsis

Although technically not part of the NDK, the STKEVENT event object is a central component to the low level architecture. It ties the HAL layer to the network scheduler thread. The network scheduler thread waits on events from various device drivers in the system including the Ethernet, serial, and timer drivers. The device drivers use the STKEVENT object to inform the scheduler that an event has occurred.

A.5.2 Object Type

Static - The STKEVENT object is created and owned by the network scheduler.

A.5.3 API Function Overview

The STKEVENT object is implemented entirely via #define MACROs and therefore, does not have a true API. This allows the network scheduler to present an abstracted API to the HAL layer for network events. The STKEVENT object is a simple structure and manipulated directly by the network control module (NETCTRL). This is discussed further in the *TMS320C6000 Network Developer's Kit (NDK) Software User's Guide* (SPRU523).

The two MACRO functions are as follows:

Property Functions:

STKEVENT_init()	Initialize a new STKEVENT object to NULL
STKEVENT_signal()	Signal a new STKEVENT event code

A.5.4 API Function Description

STKEVENT_init — ***Initialize a new STKEVENT object to NULL***

Syntax

void STKEVENT_init(STKEVENT_Handle hEvent, SEM_Handle hSem)

Parameters

hEvent	Handle to STKEVENT object
hSem	Handle to SEM object to use in STKEVENT (if any)

Return Value

None.

Description

This function is called once to initialize the STKEVENT object so it is ready for use.

Note: This function is implemented as a multi-line macro, so care should be taken when using it in the body of an if/else statement.

STKEVENT_signal *Signal a New STKEVENT Event Code*

Syntax

void STKEVENT_signal(STKEVENT_Handle hEvent, uint EventCode, uint fHwAsynch)

Parameters

hEvent	Handle to STKEVENT object
EventCode	Type of event being signaled
fHwAsynch	Flag indicating event triggered by an asynchronous hardware event (e.g., ISR, PRD).

Return Value

None.

Description

This function is called from a device driver to signal an event to the network scheduler for further processing. The STKEVENT handle *hEvent* is an event handle supplied to the device driver when the driver is first initialized. The *EventCode* parameter specifies the type of event. The currently defined events include the following:

STKEVENT_TIMER	100 ms Timer Tick Event
STKEVENT_ETHERNET	One or more Ethernet packets received
STKEVENT_SERIAL	One or more serial packets received

The *fHwAsynch* flag specifies whether the event was triggered by an external asynchronous hardware source. Examples of asynchronous events include hardware interrupts or timer PRDs. An example of a non-asynchronous event would be detecting an event from within a driver service check function. Service check functions are called periodically (or polled) by the scheduler.

Note: This function is implemented as a multi-line macro, so care should be taken when using it in the body of an if/else statement.

A.6 Link Layer Information (LLI) Object

A.6.1 Synopsis

To make full use of the stack objects described in this section, it is necessary to understand some of the stack's basic building block components. One such component is the Link Layer Information Object, or LLI for short.

An LLI object is an ARP table entry. This implementation of the IP stack combines the traditional route table and ARP table into a single table with a single API. Routes that need to use the ARP function include an ARP status object, called LLI. Normally, you only use an LLI object to inspect the ARP status of the route table.

A.6.2 Object Type

Static - LLI objects are owned and destroyed by their creator.

A.6.3 API Function Overview

The LLI API functions are as follows:

LLIGetMacAddr()	Get the Mac Address Associated with this LLI
LLIValidateRoute()	Free an LLI

A.6.4 API Functions

LLIGetMacAddr — *Get the Mac Address Associated with this LLI*

Syntax

uint LLIGetMacAddr(HANDLE hLLI, UINT8 *pMacAddr, uint MaxLen);

Parameters

hLLI	Handle to LLI object
pMacAddr	Pointer to buffer to write Mac address data
MaxLen	Maximum byte length of buffer (must be at least 6)

Return Value

Returns 1 if the Mac address for the LLI is valid and it was successfully written to the supplied buffer.

Returns 0 if the LLI does not contain a valid Mac address, or one of the calling parameters is invalid.

Description

This function is called to return the six byte Mac address associated with the LLI. It is used in system programming to obtain the hardware address from an LLI contained in a route entry.

LLIValidateRoute — *Validate an IP Address/MAC Address Pairing in the Route Table*

Syntax

HANDLE LLIValidateRoute(HANDLE hIF, IPN IPAddr, UINT8 *MacAddr);

Parameters

hIF	Handle to the interface on which the target IP address/MAC address appears
IPAddr	IP address to validate
MacAddr	Six byte MAC address corresponding to the supplied IP address

Return Value

Referenced handle to route or NULL if there was no room to create the entry.

Description

This function is called to create or update an entry in the stack route table for the supplied IP address. The entry for the given IP address is marked as valid, and assigned the supplied MAC address. Packets sent to the IP address will be assigned the given MAC address, and no ARP request will be sent.

This function also updates the route in the LLI (ARP) expiration list. It allows an application to change the state of the ARP entry even if the stack has already created the route. It should be used when it is unclear if the route (really ARP table entry) already exists or not.

Note that this function returns a referenced route handle. This handle must be dereferenced using the RtDeRef() function when it is no longer required. Because the route is treated as a standard ARP entry (with a standard expiration time as supplied in the configuration structure), the route can be dereferenced immediately.

A.7 Interface (IF) Object

A.7.1 Synopsis

The Interface (or IF) object is an abstraction of any physical interface in the system capable of transmitting and receiving packet (PKT) objects. In the current software, an interface object can represent either a PPP based device or an Ethernet (Ether) based device. However, there is no interface object, but rather PPP device objects and Ether device objects can both be treated as IF type objects for a small collection of functions. This section documents these API functions.

The IF object API covers three general areas. First, it provides a couple of generic functions to obtain information about a device, such as its type, MTU, etc.. In addition, the API also tracks physical device indices for device handles, and mapping from one to the other. This is useful for the application programming environment and configuration system, which deals in device indices instead of device handles. The last function of the IF API is to provide a generic way of creating packets for the system, keeping track of all device's header and padding requirements.

A.7.2 Object Type

Static - IF objects represent PPP or Ether objects, which are created and destroyed by the same entity.

A.7.3 API Function Overview

The following is a complete list of the IF object API. Some of these functions are only called from physical device objects like Ether or PPP.

IFInit()	Initialize handle to index mapping tables
IFIndexNew()	Allocate a new physical index for a device handle
IFIndexFree()	Free a previously allocated physical index
IFMaxIndex()	Get the highest device index currently in use
IFIndexGetHandle()	Get the device handle corresponding to a physical index
IFGetIndex()	Get a physical index corresponding to a device handle
IFGetType()	Get the interface handle type
IFGetMTU()	Get the MTU of a device
IFSetPad()	Set device header and padding requirements
IFCreatePacket()	Create a packet object for transmission

A.7.4 API Function Description

IFInit — ***Initialize Handle to Index Mapping Tables***

Syntax

void IFInit();

Return Value

None.

Description

This function is called from *ExecOpen()*, before any physical devices are initialized. It will prepare the IF system to correctly process *IFIndexNew()* commands that are called when Ether and PPP devices are created.

IFIndexNew — *Allocate a New Physical Index for a Device Handle*

Syntax uint IFIndexNew(HANDLE hIF, uint Index);

Return Value Allocated device index, or NULL on error.

Description This function is called from PPP and Ether when new physical device handles are created. IF allocates and returns a physical Index for the supplied device handle. If a specific index is required, it is passed in the *Index* parameter, otherwise *Index* is set to NULL.

IFIndexFree — *Free a Previously Allocated Physical Index*

Syntax void IFIndexFree(uint Index);

Return Value None.

Description This function is called from PPP and Ether when physical device handles are destroyed. IF frees the supplied physical Index, and can reallocate it in future calls to *IFIndexNew()*. The Index should not be used once freed.

IFMaxIndex — *Get the Highest Device Index Currently in Use*

Syntax uint IFMaxIndex();

Return Value Maximum logic device index currently in use.

Description This function returns the highest device index that is currently in use in the system. When there are no holes in the index map, this value is also the number of devices currently active.

IFIndexGetHandle — *Get the Device Handle Corresponding to a Physical Index*

Syntax HANDLE IFIndexGetHandle(uint Index);

Return Value Handle to device corresponding to supplied index, or NULL on error.

Description This function is called to convert a physical device index to a device handle.

IFGetIndex — *Get the Physical Index Corresponding to a Device Handle*

Syntax uint IFGetIndex(HANDLE hIF);

Return Value Physical device index corresponding to supplied device handle, or NULL on error.

Description This function is called to convert a device handle to a physical device index.

IFGetType — *Get the Interface Handle Type*

Syntax uint IFGetType(HANDLE hIF);

Return Value Handle type of supplied handle.

Description This function is called to get the handle type of the supplied device handle. When called correctly, the return value should be one of the following:

HTYPE_ETH	Ether Device
HTYPE_PPP	PPP Device

IFGetMTU — *Get the MTU of a Device*

Syntax uint IFGetMTU(HANDLE hIF);

Return Value MTU of the device indicated by the supplied handle.

Description This function is called to get the MTU (maximum transmit unit) size of the indicated device. The MTU value does not include the device's layer 2 header. Thus, for Ethernet and serial PPP, the MTU will normally be 1500; however, for protocols like PPPoE, the MTU will be smaller.

IFSetPad — *Set Device Header and Padding Requirements*

Syntax void IFSetPad(uint Header, uint Padding);

Return Value None.

Description This function is called by a physical device object to set the layer 2 header and padding requirements for a packet. For example, with Ethernet, the header is normally 14. Plus, if the Ethernet checksum appears in the packet body, the value of padding is 4.

IFCreatePacket — *Create a Packet Object for Transmission*

Syntax HANDLE IFCreatePacket(uint size);

Return Value Handle to new packet buffer (PBM), or NULL on allocation error.

Description This function is probably the most useful of the IF functions. It is called to create a packet object to send packets out of the stack. It uses information collected from the physical devices to create a packet that can be transmitted on any of the physical devices in the system. It does this by applying worst case header and padding sizes. The handle returned by this function references a packet buffer created by the packet buffer manager (PBM). The packet buffer object is described in Section A.3. This function is preferred over calling *PBM_alloc()* because it sets up the packet for use by the stack. The data offset property is set to where the IP header should be placed. This offset guarantees that the packet can be transmitted on any packet device in the system.

A.8 Ether Object

A.8.1 Synopsis

The Ether object is really just the generic portion of the packet driver. It knows how to process an Ethernet MAC header, and handles incoming and outgoing packets. It interfaces directly to the HAL packet driver. For each Ethernet based packet device in the system, an Ether object is created to represent this device to the stack.

A.8.2 Object Type

Static - Ether objects are generally created and destroyed by the same entity.

A.8.3 API Function Overview

The following is a complete list of the Ether object API.

Create/Destroy Functions:

EtherNew()	Create New Ether Object
EtherFree()	Destroy Ether Object
EtherConfig()	Configure Ether Object Header Parameters

Addressing Functions:

EtherGetMacAddr()	Get the Device's Unicast MAC Address
EtherAddMCast()	Add Multicast Ethernet Address
EtherDelMCast()	Delete Multicast Ethernet Address
EtherClearMCast()	Clear All Multicast Ethernet Addresses

Filtering Functions:

EtherSetPktFilter()	Set Receive Packet Filter Value
EtherGetPktFilter()	Get Current Receive Packet Filter Value

Hardware Event Functions:

EtherRxPacket()	Indicate a New Rx Packet to the Ether Object

A.8.4 API Functions

Although the Ether object API is larger than that discussed here, this section covers the portion of the API that is useful to a system application.

EtherNew — ***Create New Ether Object***

Syntax

HANDLE EtherNew(uint PhysIndex);

Return Value

Returns a handle to the Ether object, or NULL on a memory allocation error.

Description

Installs a new Ether object in the system. This call should be made for every ethernet device installed. Once called, the stack will make calls to the HAL packet driver interface to get more information about the device. The argument is the physical device id used by the HAL to identify the device.

EtherFree — ***Destroy Ether Object***

Syntax

void EtherFree(HANDLE hEther);

Description

Destroys the indicated Ether object, and frees its associated memory. This function should be called to remove devices after the stack has shut down. Calling this function will not result in any calls to the HAL.

EtherConfig — ***Configure Ether Object***

Syntax

void EtherConfig(HANDLE hEther, uint PhysMTU, uint EthHdrSize, uint OffDstMac, uint OffSrcMac, uint OffEthType, uint PacketPad);

Description

Describes to the Ether object how the Ethernet header is constructed on this device. Although the MAC address is assumed to be 6 bytes long, various devices have a small variety of packet variances. The Ether device object must know this information to both process and construct packets in buffers that are native to the physical device.

The arguments are defined as follows:

PhysMTU	Physical MTU of the packet (usually 1514)
EthHdrSize	Minimum (non-802.2 SNAP) header size (usually 14)
OffDstMac	Byte offset from header start to DST Mac Addr (usually 0)
OffSrcMax	Byte offset from header start to Src Mac Addr (usually 6)
OffEthType	Byte offset from header start to ether type (usually 12)

PacketPad	Required byte pad at end of frame (usually 0 or 4)

EtherGetMacAddr — *Get the Device's Unicast MAC Address*

Syntax

uint EtherGetMacAddr(HANDLE hEther, UINT8 *pMacAddr, uint MaxLen);

Description

Called to retrieve the unicast MAC address of the physical Ethernet device. The MAC address is written to the pointer *pMacAddr*. The maximum length of the buffer must be at least 6 bytes and is specified in MaxLen. The function returns 1 on success and 0 on failure.

EtherAddMCast — *Add Multicast Ethernet Address*

Syntax

uint EtherAddMCast(HANDLE hEther, UINT8 *pMCastAddr);

Description

Called to add an Ethernet multicast address to the list of addresses to be received by the Ethernet hardware when the Rx filter is set to ETH_PKTFLT_MULTICAST. The multicast address is specified by the pointer *pMCastAddr*, pointing to a six byte MAC address. The multicast address list can also be manipulated in its raw form at the *llPacket* layer (see Section D.4).

EtherDelMCast — *Delete Multicast Ethernet Address*

Syntax

uint EtherDelMCast(HANDLE hEther, UINT8 *pMCastAddr);

Description

Called to remove an Ethernet multicast address from the list of multicast addresses previously added via a call to *EtherAddMCast()*. The multicast address to remove is specified by the pointer *pMCastAddr*, pointing to a size byte MAC address. The multicast address list can also be manipulated in its raw form at the *llPacket* layer (see Section D.4).

EtherClearMCast — *Clear All Multicast Ethernet Addresses*

Syntax

void EtherClearMCast(HANDLE hEther);

Description

Called to remove all Ethernet multicast addresses from the list of multicast addresses previously added via a call to *EtherAddMCast()*. After calling this function, the Ethernet adapter will not receive any multicast addresses when the Rx filter is set to ETH_PKTFLT_MULTICAST or below. The multicast address list can also be manipulated in its raw form at the *llPacket* layer (see Section D.4).

EtherSetPktFilter — *Set Receive Packet Filter Value*

Syntax

void EtherSetPktFilter(HANDLE hEther, uint PktFilter);

Description

Called to indicate the level of filtering for Ethernet packets. By default, the driver is opened with filter value: ETH_PKTFLT_MULTICAST. Valid filter values are as follows:

ETH_PKTFLT_NOTHING	No Packets
ETH_PKTFLT_DIRECT	Only directed Ethernet
ETH_PKTFLT_BROADCAST	Directed plus Ethernet Broadcast
ETH_PKTFLT_MULTICAST	Directed, Broadcast, and selected Ethernet Multicast
ETH_PKTFLT_ALLMULTICAST	Directed, Broadcast, and all Multicast
ETH_PKTFLT_ALL	All packets

For selecting multicast addresses as the ETH_PKTFLT_MULTICAST level, see *EtherAddMCast()*.

EtherGetPktFilter — *Get Current Receive Packet Filter Value*

Syntax uint EtherGetPktFilter(HANDLE hEther);

Description Called to retrieve the current level of filtering for Ethernet packets. See the description of *EtherSetPktFilter()* for more information.

EtherRxPacket — *Indicate a New Rx Packet to the Ether Object*

Syntax void EtherRxPacket(PBM_Handle hPkt);

Description Called to indicate the reception of a new packet to the corresponding Ether object. The Ether object takes ownership of the indicated packet buffer, until it is returned via a call to the packet buffer manager (PBM).

The argument hPkt is the handle of a standard packet buffer object. The valid data, offset, and receiving interface fields must be valid. The packet buffer object is described in Section A.3.

A.9 Binding Object

A.9.1 Synopsis

For a device object to live on the network, it must have an IP address and knowledge of its IP subnet. The process of assigning an IP address and subnet to a device binds the device with the desired IP addressing.

A.9.2 Object Type

Static - Binding objects are generally created and destroyed by the same entity.

A.9.3 BIND API Functions

Although the Bind object API is larger than that discussed here, this section covers the portion of the API that is encountered by a system application.

BindNew — *Create New IP Binding*

Syntax HANDLE BindNew(HANDLE hIF, IPN IPAddr, IPN IPMask);

Return Value Returns a handle to the Bind object, or NULL on error.

Description Binds the indicated IP address and mask to the supplied Ether device. The handle to the Ether device object is specified as hIF - or an handle to an *interface*, because the interface may or may not be an Ethernet device (but always is in this version).

The IP address and mask arguments are given the type IPN, which is an unsigned 32 bit value. IPN stands for IP Network format, meaning that the IP data must be supplied in network format. If unsure of the network format for your hardware, use the htonl() macro function on the native format (where 1.2.3.4 = = 0x01020304).

BindFree — *Destroy IP Binding Object*

Syntax void BindFree(HANDLE hBind);

Description Destroys the indicated Bind object, and frees its associated memory. This function removes the IP address and subnet association in the system route table. It has no effect on the Ether object involved in the binding.

BindGetFirst — *Start Enumeration of Binding Objects*

Syntax HANDLE BindGetFirst();

Description Returns a handle to the first binding installed in the system (or NULL if no bindings exist).

BindGetNext — *Continue Enumeration of Binding Objects*

Syntax HANDLE BindGetNext(HANDLE hBind);

Description Returns a handle to the binding in the installed binding list that follows the indicated binding (or NULL if no more bindings exist). Note that bindings are not internally kept in chronological order in which they were installed.

BindGetIF — *Get the Ether Object that is Bound by this Binding Object*

Syntax HANDLE BindGetIF(HANDLE hBind);

Description Returns a handle to the Ether object that is bound by this binding object. Note that a binding is nothing more than an assignment of an Ether object to an IP address/network.

BindGetIP — *Get the IP Address/Network that is Bound by this Binding Object*

Syntax void BindGetIP(HANDLE hBind, IPN *pIPHost, IPN *pIPNet, IPN *pIPMask);

Description Returns the IP address and mask as requested by the calling arguments. Any of the pointer arguments can be NULL if the information is not required.

The arguments are defined as follows:

pIPHost	Pointer to the local IP address assigned by this binding
pIPNet	Pointer to the network assigned by this binding (IP address AND IP Mask)
pIPMask	Pointer to the subnet mask of the network assigned by this binding

A.10 Route Object

A.10.1 Synopsis

The route manager maintains IP routing information. It is called by various routines to get and set route information. A route object is a destination on the network. Locally, it consists of an egress interface and a next hop IP address.

This section describes a subset of the route object. Flags, features, and API calls have been omitted for simplicity. Also, documenting the entire API would require the documentation of other stack objects that are not covered in this document.

A.10.2 Object Type

Referenced - Route objects are referenced and dereferenced as needed. The object is removed when the reference count reaches ZERO.

A.10.3 Route Entry Flags Definition

Associated with each route is a collection of entry/status flags. These flags indicate the type of route and its status. Most system programming is not concerned with the route entry flags. They are listed here for completeness. The definition of the various flags is as follows:

- FLG_RTE_UP - Entry is up

  When set, indicates that the route is valid. The only time this flag is cleared is when the route is being initialized, or when an error condition is signaled via RtSetFailure(). The flag is reset to TRUE by calling RtSetFailure() with NULL failure code, or if the route is modified.
- FLG_RTE_EXPIRED - Entry is expired

  When set, indicates that the route is expired. The flag cannot be cleared. A new route must be created. Expired routes are never found, but a route cached by another entity may expire while it is being held.
- FLG_RTE_STATIC - Entry is static

  This flag is set when a route should remain in the routing table even if it has no references. Various routes can be static. Static routes are manually referenced by the system during create, and manually de-referenced by the system during system shutdown.
- FLG_RTE_BLACKHOLE - Entry is a blackhole

  When set, indicates that the route is a black hole. All packets destined for this address are silently discarded.
- FLG_RTE_REJECT - Entry is rejected

  When set, indicates that the route is to an invalid address. All packets destined for this address are discarded with an error indication.
- FLG_RTE_MODIFIED - Route has been auto modified

  When set, indicates that the route has been modified as a result of an ICMP redirect message. This can occur only to GATEWAY routes, and only if ICMP modifications are enabled in the stack configuration.
- FLG_RTE_DYNAMIC - Route has been auto created

  When set, indicates that the route has been created as a result of an ICMP redirect message. ICMP can only create GATEWAY routes, and may do so only if ICMP modifications are enabled in the stack configuration.
- FLG_RTE_PROXYPUB - Reply to ARP with client's MAC address

  This flag indicates that the router is a proxy publisher of another entity's MAC address. When set, the ARP protocol will respond to ARP requests for the route's IP address with the supplied static MAC address when the host is on the same IF device as the incoming ARP request. This allows support of hosts that do not implement ARP but are on the same physical Ethernet network. PROXYPUB entries are always created with a MAC address and contain a static LLI (link-layer info, i.e., ARP entry).

?. FLG_RTE_PROXY - Reply to ARP with router's MAC address

  This flag indicates that the router is acting as a proxy for this host or network route. When set, the ARP protocol will respond to ARP requests with its own MAC address for the associated IP host or network when the network appears on a different IF device from the incoming ARP request. The MAC address supplied in the reply is the local MAC of the ingress IF device. This technique tricks clients into sending packets to the router when subnets are split across physical devices on a router.

  One potential use applies when the stack is acting as a PPP server and Ethernet router. If a PPP client is made part of the same IP subnet as an Ethernet based interface, the stack acts as the PPP client's proxy so that Ethernet peers can communicate via ARP.
- FLG_RTE_CLONING - Cloning route to a local IP subnet

  When set, indicates that the network route is a cloning route. Cloning routes clone (spawn to) host routes when a route search is performed on a host address that is a member of the cloning route's network (via the address and subnet mask). Cloned host routes take on most of the properties of their parent network route, with the following alterations:
  - Any MODIFIED or DYNAMIC flags are cleared.
  - The STATIC flag is never set.
  - The HOST flag is set and the netmask is set to 1s.
  - The CLONING flag is cleared.

Note: Cloning routes are routes to a network (IP and subnet). These routes are added automatically when an IP network is added to a device via a Bind object. Take care when adding this type of route manually.

- FLG_RTE_HOST - Host route (no subnet mask)
  When set, indicates that the route entry is a host route. A host route has no subnet mask (or rather a subnet mask of all 1's). When searching for a route, host routes always match before network routes (but this behavior can be overridden).
- FLG_RTE_GATEWAY - Destination is available via a Gateway
  When set, indicates that the host or network route is indirectly accessible via an IP gateway. For a route with this flag set, the GateIP address is always valid. Most GATEWAY routes will also be network routes; however, a host redirect from ICMP can create a host route with a different gateway than its parent route. When searching for a route, gateway routes always match before host routes (but this behavior can be overridden).
- FLG_RTE_IFLOCAL - IP address is Local to the stack
  When set, indicates that the host route does not have a valid LLI (ARP) entry because the host is local to the stack. The MAC address of this local IP host address can be obtained from the interface handle associated with the route.

Note: Local routes are in the routing table to route packets that originate in the stack's upper layers. When handling ARP requests and routing of incoming packets from outside the stack, the IP address list published via the Bind object is used. The ARP will not respond to, nor will the IP accept, packets addressed to an IP address that is not in the Bind list, even if an IFLOCAL address entry exists in the route table. As with a cloning route, the Bind object is the best way to create a local route.

A.10.4 Route Entry Flags Guidelines

See the following for some general guidelines to use when creating new routes. Use the definitions listed above with the following legal flag combinations:

- Setting FLG_RTE_BLACKHOLE
  FLG_RTE_REJECT - must be OFF
- Setting FLG_RTE_REJECT
  FLG_RTE_BLACKHOLE - must be OFF
- Setting FLG_RTE_CLONING
  FLG_RTE_HOST - must be OFF
  FLG_RTE_GATEWAY - must be OFF
  FLG_RTE_IFLOCAL - must be OFF
- Setting FLG_RTE_HOST
  FLG_RTE_CLONING - must be OFF
- Setting FLG_RTE_GATEWAY
  FLG_RTE_CLONING - must be OFF
  FLG_RTE_IFLOCAL - must be OFF
- Setting FLG_RTE_IFLOCAL
  FLG_RTE_HOST - must be ON
  FLG_RTE_CLONING - must be OFF
  FLG_RTE_GATEWAY - must be OFF
- Setting FLG_RTE_PROXYPUB
  FLG_RTE_HOST - must be ON
  FLG_RTE_CLONING - must be OFF
  FLG_RTE_GATEWAY - must be OFF
- Setting FLG_RTE_PROXY
  FLG_RTE_CLONING - must be OFF
  FLG_RTE_GATEWAY - must be OFF

A.10.5 API Functions

The Route API is the most extensive API that a system task uses outside of the stack routines themselves. As with the other stack APIs, this guide does not document the entire API.

Calls that accept a CallFlags argument can be supplied with the FLG_RTF_REPORT flag to indicate that the call should result in a route report to the route control object. The route control object is described later in this section.

RtRef — *Reference a Route*

Syntax

void RtRef(HANDLE hRt);

Description

Called to add one to the reference count of a route. An application that keeps a route it did not create itself should reference the route before it uses it, and dereference it when it is through.

RtDeRef — *Dereference a Route*

Syntax

void RtDeRef(HANDLE hRt);

Description

Called to remove one from the reference count of a route. An application dereferences a route when it is through with it. This is the same (to the application) as destroying the route. The route is actually destroyed when its reference count reaches zero.

RtCreate — *Create New Route*

Syntax

HANDLE RtCreate(uint CallFlags, uint RtFlags, IPN IPAddr, IPN IPMask, HANDLE hIF, IPN IPGateway, UINT8 *pMacAddr);

Parameters

CallFlags	Call Type Flags
RtFlags	Route Type Flags
IPAddr	Destination IP address of route
IPMask	Destination IP Mask of route (or NULL)
hIF	Interface (or NULL)
IPGateway	Gate IP address (or NULL)
pMacAddr	Pointer to six byte MAC address (or NULL)

Call Flags

FLG_RTF_REPORT	Reports new route (NEW)

Return Value

Referenced handle to newly created route.

Description

Called to create a new host or network route and add it to the route table. Existing routes cannot be modified via this call.

Some flag combinations are incorrect, and the following rules are strictly enforced.

- FLG_RTE_UP flag is always SET.
- FLG_RTE_EXPIRED and FLG_RTE_MODIFIED flags are always CLEARED.
- If FLG_RTE_HOST is set, then the route is a host route and *IPMask* is ignored, and FLG_RTE_CLONING cannot be set.
- If FLG_RTE_GATEWAY is set, then *IPGateway* must specify a valid (reachable) IP address.
- If FLG_RTE_GATEWAY is not set, then *hIF* must be valid.
- If FLG_RTE_IFLOCAL is set, then the specified host address is local to this machine,

and FLG_RTE_HOST must also be set, FLG_RTE_GATEWAY cannot be set, and *hIF* must be valid.

- If FLG_RTE_CLONING is specified in Flags, the route is a cloning network route. The *IPMask* argument must be valid, and neither FLG_RTE_HOST nor FLG_RTE_GATEWAY may be set.
- If FLG_RTE_STATIC is specified in Flags, the route is referenced once by the route code, and later dereferenced during shut down.

RtFind — *Find a Route*

Syntax

void RtFind(uint CallFlags, IPN IPAddr);

Call Flags

FLG_RTF_REPORT	Reports any new (cloned) or unfound route (NEW or MISS)

Return Value

Referenced handle to best match route (or NULL)

Description

This call searches the route table for a route that matches the supplied IP address. The search always returns the best match for a route. The best match is a match with the most bits in the subnet mask. Thus, a host match takes priority over a network match.

When there is more than one route with the same subnet mask, the following matching guidelines are used (listed from best to worst):

- Route has a local destination (occurs with host addresses only).
- Route has a gateway destination.
- Route has a subnet destination on a connected interface.

Sometimes a search is desired where particular matches are desired. The following flags can be combined with the value of *CallFlags* to change the behavior of the search:

FLG_RTF_CLONE	Clone a network route to a host route if host not found
FLG_RTF_HOST	Find only non-gateway host routes

RtSetTimeout — *Set the Timeout for a Non-static Route*

Syntax

void RtSetTimeout(HANLE hRt, UINT32 dwTimeOut);

Description

This call allows an application to specify that the stack should time out a referenced route. When the route is added to the timeout list, the system will add a reference. Thus, once the application sets the timeout value, it should call *RtDeRef()* to dereference the route. The route will stay valid until the timeout value is exceeded, after which it is dereferenced by the system. Note that if this function is called and the route is not dereferenced by the caller, it will still be removed from the system route table when the expiration time elapses, but the object will not be freed.

RtSetFailure — *Set the Timeout for a Non-Static Route*

Syntax

void RtSetFailure(HANLE hRt, uint CallFlags, uint FailCode);

Call Flags

FLG_RTF_REPORT	Reports the status change of the route (UP or DOWN)

Description

This call allows an application to specify a particular error with a route, or clear a previously indicated error. Setting an error clears the FLG_RTE_UP bit in the flags. When use of the route is attempted, the specified error is returned. Defined error codes for the *FailCode* argument are:

NULL	Route is operating normally (sets FLG_RTE_UP flag)
RTC_HOSTDOWN	Host is down
RTC_HOSTUNREACH	Host unreachable
RTC_NETUNREACH	Network unreachable

RtRemove — *Remove Route from System Route Table*

Syntax

void RtRemove(HANLE hRt, uint CallFlags, uint FailCode);

Call Flags

FLG_RTF_REPORT	Reports the removal of the route (REMOVED)

Description

This call allows an application to remove a route from the system route table independently of any held references to the route. It is similar to the *RtSetFailure()* call, but differs in two ways:

1. It removes the route from the system route table so that it can no longer be returned by *RtFind()*.
2. It calls the IP and Sockets layers to flush the route from any local cache.

Calling this function clears the FLG_RTE_UP bit in the flags. When use of the route is attempted, the error specified in *FailCode* is returned. Defined error codes for the *FailCode* argument are:

RTC_HOSTDOWN	Host is down
RTC_HOSTUNREACH	Host unreachable
RTC_NETUNREACH	Network unreachable

RtGetFailure — *Set the Timeout for a Non-Static Route*

Syntax

uint RtGetFailure(HANLE hRt);

Return Value

Failure code or NULL for normal operation.

Description

This call allows an application to retrieve the error code of a route where the FLG_RTE_UP bit is not set in the route flags. Defined error codes are:

RTC_HOSTDOWN	Host is down
RTC_HOSTUNREACH	Host unreachable
RTC_NETUNREACH	Network unreachable

RtGetFlags — *Get the Route Flags*

Syntax uint RtGetFlags(HANLE hRt);

Description This function returns the state of the route flags for the indicated route. The flag values and definitions were discussed earlier in this section.

RtGetIPAddr — *Get the Route IP Address*

Syntax IPN RtGetIPAddr(HANLE hRt);

Return Value IP host/network address.

Description This function returns the specified route's IP address in network format.

RtGetIPMask — *Get the Route IP Subnet Mask*

Syntax IPN RtGetIPMask(HANLE hRt);

Return Value IP subnet mask.

Description This function returns the specified route's IP subnet mask in network format.

RtGetGateIP — *Get the Route Gateway IP Address*

Syntax IPN RtGetGateIP(HANLE hRt);

Return Value IP address of the Gateway or NULL.

Description This function returns the Gateway IP address for the specified route (assuming the FLG_RTF_GATEWAY bit is set in the route flags).

RtGetIF — *Get the Route's Destination Hardware Interface*

Syntax HANDLE RtGetIF(HANLE hRt);

Return Value HANDLE to Ether Object representing target interface.

Description This function returns an Ether device handle to the egress (target) device of the route. Even local IP addresses have target devices (the device they are bound to).

RtGetMTU — *Get the MTU of a Packet Sent via this Route*

Syntax uint RtGetMTU(HANLE hRt);

Return Value Packet payload MTU in bytes.

Description This function returns the MTU (not including layer 2 header) of a packet sent via the supplied route.

RtWalkBegin — *Start Walking the Route Table*

Syntax HANDLE RtWalkBegin();

Return Value HANDLE to first route in system route table or NULL if no routes.

Description This function initiates a walk of the route table. It returns the first route in the table. The walk must be terminated with *RtWalkEnd()* for the system to behave properly.

RtWalkNext — *Get Next Route While Walking the Route Table*

Syntax HANDLE RtWalkNext(HANDLE hRt);

Return Value HANDLE to next route in system route table or NULL if no routes.

Description This function gets the next route (based off the previous route supplied) in a walk of the route table. The walk must be terminated with *RtWalkEnd()* for the system to behave properly.

RtWalkEnd *Stop Walking the Route Table*

Syntax void RtWalkEnd(HANDLE hRt);

Description This function completes the walk of the route table. The last route (if any) obtained from RtWalkBegin() or RtWalkNext() is specified in the calling argument. Otherwise, NULL is used.

A.11 Route Control Object

A.11.1 8.12.1 Synopsis

The route control object is more of a function than an object. It serves as a collection point for route related information in the system. A routing daemon may use this information, or it could simply be logged as debugging information.

When so configured, route control messages are transformed into debug messages by the stack and logged via *DbgPrintf()*. By default, the route control debug messages are disabled. Also, the message function can be hooked by an application.

Note, control messages can also be suppressed individually by not supplying the FLG_RTF_REPORT flag to the Route object API function when the call is made (as mentioned in the previous section).

A.11.2 Route Control Messages

The basic form of the route control message is an unsigned int message value, with two unsigned 32 bit values for additional data. In most cases these are immediate data. In one instance, the value is actually a 32 bit memory pointer.

Messages are passed internally to the stack via the function:

```
void  RTCReport(uint Msg, UINT32 Param1, UINT32 Param2);
```

Applications should not call this function directly.

The possible values for *Msg* are as follows:

MSG_RTC_UP *Route is Valid/Pending*

Parameters

Param1	Route IP
Param2	Route IP Mask (all ones for host route)

Description Called after a down message indicating that a route that had previously been in the down state is now up again. This does not mean that the route has been validated, but only that it will attempt to validate itself if used.

MSG_RTC_DOWN *Route is Down*

Parameters

Param1	Route IP
Param2	Route IP Mask (all ones for host route)

Description Called when a route goes down due to an error. Packets sent via a route in this state will generate an error. The most common reason for a route to go down is for a non-response to 5 successive ARP requests. In this case, the route will come back up after the down time has expired.

MSG_RTC_MISS *Route Find "Missed" on Route*

Parameters

Param1	Route IP
Param2	Route IP Mask (all ones for host route)

Description Called when the route table was searched for a route and no matching route was found. This message will never be sent when there is a default route in the table because all searches will have a match (unless a special restricted search is performed).

MSG_RTC_NEW *New Route has been Entered into the Route Table*

Parameters

Param1	Route IP
Param2	Route IP Mask (all ones for host route)

Description Called when a new route is created and entered into the route table. Routes can be created by applications, when new bindings are created, by ICMP redirects, or when local host routes are cloned from local subnet routes.

MSG_RTC_EXPIRED *Route has Expired*

Parameters

Param1	Route IP
Param2	Route IP Mask (all ones for host route)

Description Called when a route with an expiration timeout has expired and been removed from the table.

MSG_RTC_REMOVED *Route has been Manually Removed*

Parameters

Param1	Route IP
Param2	Route IP Mask (all ones for host route)

Description Called when a route has been manually removed from the table. This message is not generated when static routes are removed at system shutdown. Generally, a route can only be removed when its reference count reaches zero. This cannot happen to a static route or a route with an expiration timeout. For the former, no message is ever generated. For the latter, the MSG_RTC_EXPIRED message is used.

MSG_RTC_MODIFIED *Route has been Manually Modified*

Parameters

Param1	Route IP
Param2	Route IP Mask (all ones for host route)

Description Called when a route has been manually modified via the RtModify() call. The stack does not use this function, so if it is not called by an application, this message will never occur.

MSG_RTC_REDIRECT *Route has been Redirected*

Parameters

Param1	Route IP
Param2	New Destination Gateway IP

Description Called when an ICMP redirect message is received for a given IP host address. Because the invention of classless subnets, all redirects are treated as HOST redirects. If the stack is configured to generate redirect routes automatically (will do so by default), this message will occur after the new static host redirect route has been created (which will also generate a MSG_RTC_NEW message). If the stack does not create the redirect route, this message occurs *before* the socket layer is notified so that if a new route is created as a result of this message, the sockets layer will find it.

MSG_RTC_DUPIP *A Duplicate IP Address has been Detected in the System*

Parameters

Param1	Duplicated IP
Param2	Pointer to 6 byte MAC address of offending device

Description Called when an ARP packet is received from a device that has an IP address that is the same as the IP address of the stack on that physical interface. Depending on the age of the address, the application may wish to destroy the binding.

A.11.3 Route Control API Functions

RTCAddHook — *Hook RTC Messages*

Syntax uint RTCAddHook (void (*pfn)(uint, UINT32, UINT32));

Return Value 1 if the hook was installed, or NULL on an error (too many hooks).

Description Called to hook a message function to receive route control messages. The argument is a pointer to a message function of the type:

```
void  MyMsgFun(uint Msg, UINT32 Param1, UINT32 Param2);
```

Note that the supplied callback function is called from within an *llExit()*/*llEnter()* pair, and thus may call the stack API directly, but may not call any applications API functions, like sockets functions. If such action is required, the callback function may call *llExit()* when called and then *llEnter()* before returning.

When the hook is no longer required, the function may be unhooked by calling *RTCRemoveHook()*.

RTCRemoveHook — *Unhook RTC Messages*

Syntax void RTCRemoveHook (void (*pfn)(uint, UINT32, UINT32));

Return Value None.

Description Called to remove a previously hooked callback function.

A.12 Configuring the Stack

A.12.1 Synopsis

The stack has multiple configuration options that can be changed by the system programmer. This is possible by altering the default values in a stack configuration structure before the stack is initialized.

A.12.2 Configuration Structure

The stack internal configuration structure is _ipcfg. Any element in this structure may be modified before the initial system call to ExecOpen(). This structure should not be modified after this initial call.

The _ipcfg structure is of type IPCONFIG, which is defined as follows:

```
typedef struct _ipconfig {
    uint    IcmpDoRedirect;     // Update route table on ICMP redirect
    uint    IcmpTtl;            // TTL for ICMP messages (RFC1700 says 64)
    uint    IcmpTtlEcho;        // TTL for ICMP echo (RFC1700 says 64)
    uint    IpIndex;            // IP Start Index
    uint    IpForwarding;       // IP Forwarding (1 = Enabled)
    uint    IpNatEnable;        // IP NAT Enable (1 = Yes)
    uint    IpFilterEnable;     // IP Filtering Enable (1 = Yes)
    uint    IpReasmMaxTime;     // Max reassembly time in seconds
    uint    IpReasmMaxSize;     // Max reassembly packet size
    uint    IpDirectedBCast;    // Look for directed Broadcast IP addresses
    uint    TcpReasmMaxPkt;     // Max reassembly pkts held by TCP socket
    uint    RtcEnableDebug;     // Enable Route Control Messages (1 = On)
    uint    RtcAdvTime;         // Time in seconds to send Router Advertisments (0 = don't)
    uint    RtcAdvLife;         // Litetime of route in Router Advertisments
    int     RtcAdvPref;         // Preference Level (signed) in Router Advertisments
    uint    RtArpDownTime;      // Time 5 failed ARPs keep Route down (sec)
    uint    RtKeepaliveTime;    // VALIDATED route timeout (sec)
    uint    RtCloneTimeout;     // INITIAL route timeout (sec)
    uint    RtDefaultMTU;       // Default MTU for internal routes
    uint    SockTtlDefault;     // Default Packet TTL
```

```
    uint    SockTosDefault;     // Default Packet TOS
    int     SockMaxConnect;     // Max Socket Connections
    uint    SockTimeConnect;    // Max time to connect (sec)
    uint    SockTimeIo;         // Default Socket IO timeout (sec)
    int     SockTcpTxBufSize;   // TCP Transmit buffer size
    int     SockTcpRxBufSize;   // TCP Receive buffer size (copy mode)
    int     SockTcpRxLimit;     // TCP Receive limit (non-copy mode)
    int     SockUdpRxLimit;     // UDP Receive limit
    int     SockBufMinTx;       // Min Tx space for "able to write"
    int     SockBufMinRx;       // Min Rx data for "able to read"
    uint    PipeTimeIo;         // Default Pipe IO timeout (sec)
    int     PipeBufSize;        // Pipe internal buffer size
    int     PipeBufMinTx;       // Min Tx space for "able to write"
    int     PipeBufMinRx;       // Min Rx data for "able to read"
    uint    TcpKeepIdle;        // Keep idle time (0.1 sec units)
    uint    TcpKeepIntvl;       // Keep probe interval (0.1 sec units)
    uint    TcpKeepMaxIdle;     // Keep probe timeout (0.1 sec units)
    uint    IcmpDontReplyBCast; // Do NOT reply to ICMP Echo Request packets sent to broadcast
                                // or directed broadcast addresses.
    uint    IcmpDontReplyMCast; // Do NOT reply to ICMP Echo Request packets sent to
                                // multicast broadcast addresses.

} IPCONFIG;;
```

The structure entries are defined as follows:

_ipcfg.IcmpDoRedirect *Update Route Table on ICMP Redirect*

Default Value 1 (Yes)

Description When set, causes ICMP to automatically create a route to perform redirects on an IP host to the gateway supplied in the redirect message. If set to false (0), you can take whatever action you feel necessary as the ICMP redirect will also generate a route control message.

_ipcfg.IcmpTtl *TTL for ICMP Messages*

Default Value 64

Description This is the TTL value ICMP will use in messages it generates as a result of routing IP packets. Legal values are in the range of (1-255).

_ipcfg.IcmpTtlEcho *TTL for ICMP ECHO Reply Messages*

Default Value 255

Description This is the TTL value ICMP will use in echo reply messages it generates in response to receiving echo requests. Legal values are in the range of (1-255).

_ipcfg.IpIndexStart *IP Start Index*

Default Value 1

Description This is the initial value that is placed in the IP Id field for IP packets generated by the system. Legal values are in the range of (1-65535).

_ipcfg.IpForwarding *IP Forwarding Enable*

Default Value 0 (No)

Description When set to true (1), this allows the stack to forward packets it receives for other IP address to their next hop destination (i.e., it allows the stack to act as a router).

_ipcfg.IpNatEnable *IP Network Address Translation Enable*

Default Value 0 (No)

Description When set to true (1), this allows the stack to make use of the network address translation (NAT) module. Note that in addition to setting this structure element, NAT must also be configured. This is described in the following section.

_ipcfg.IpReasmMaxTime *Maximum IP Packet Reassembly Time in Seconds*

Default Value 10

Description This is the maximum time that the stack will hold IP packet fragments while attempting to assemble a complete packet. If the time expires before all the fragments arrive, the packet is discarded.

_ipcfg.IpReasmMaxSize *Maximum IP Packet Reassembly Packet Size in Bytes*

Default Value 3020

Description This is the maximum packet size that the stack will attempt to reassemble. As soon as the stack determines that the total packet size exceeds this value, the packet is discarded. The default size of 3020 is the maximum size given the default implementation of the packet buffer manager (PBM). If a larger size is desired, then large buffer support must be added to the PBM module. This value is not otherwise restricted. Note the MAC and IP headers are not included in this size limit.

_ipcfg.IpDirectedBCast *Look for Directed Broadcast IP Packets*

Default Value 1 (Yes)

Description A directed broadcast address is one where all the bits in the subnet portion of the address are set to 1. For example, on the network 192.168.1.0:255.2555.255.0, the IP address 192.168.1.255 would be a directed broadcast IP. This address is treated as a broadcast for both IP send and receive. The IP layer can be told to disable directed broadcast by setting this value to zero. When disabled, the directed broadcast address is treated like any other host address.

_ipcfg.TcpReasmMaxPkt *Maximum Reassembly Packets Held by TCP Socket*

Default Value 2

Description The TCP layer has its own packet reassembly module, allowing TCP packets to arrive out of order, and yet be properly reassembled without the need to retransmit data. One potential issue with embedded environments where the socket receive buffers are large is that a significant number of packets can be tied up in TCP if the first packet of a large burst is lost. This value allows you to specify the maximum number of packets the TCP layer will hold per socket pending reassembly, or in other words, the maximum number of out of order packets allowed.

_ipcfg.RtcEnableDebug *Enable Route Control Messages*

Default Value 0 (No)

Description Route control messages keep the system informed of route updates. When set to Yes (1), this variable causes RTC to process the route control message and convert the message into a debug call to *llDebugMessage()*. Note that an application may also hook into the RTC message loop using the *RTCAddHook ()* function.

_ipcfg.RtcAdvTime *Time in Seconds to Send Router Advertisments*

Default Value 0 (Do not Send Router Advertisements)

Description The stack has the ability to automatically send ICMP router advertisements at a predetermined interval. Setting this variable to a non-zero value determines the interval.

_ipcfg.RtcAdvLife *Lifetime of Route in Router Advertisments*

Default Value 120

Description If sending router advertisements (see above), this is the route lifetime that will be sent in the ICMP message.

_ipcfg.RtcAdvPref *Preference Level of Route in Router Advertisments*

Default Value 0

Description If sending router advertisements (see above), this is the route preference level that will be sent in the ICMP message. This value is signed.

_ipcfg.RtDownTime *Time in Seconds a Route is "Down" Due to Failed ARP*

Default Value 20

Description To stop an application from sending endless packets to a route that is not responding to ARP, the route is brought down for a period of time so that the application will receive an error when IP attempts to send. After the designated time, the route is brought back up and will attempt more ARP requests if used again.

_ipcfg.RtKeepAliveTime *Time in Seconds a Validated Route is Held*

Default Value 1200

Description Routes should not be held indefinitely. Use of a route is also not sufficient to keep the route alive. This value represents the time an ARP validated route is held before it expires. If the route is revalidated via ARP during this period, the period is extended for this interval from that point in time.

_ipcfg.RtCloneTimeout *Default Timeout in Seconds of a Cloned Route*

Default Value 120

Description When a host route is first cloned from a network route, it is assigned this default timeout. Once the route is validated via ARP, the timeout is extended (see above).

_ipcfg.RtDefaultMTU *Default MTU for Local Routes*

Default Value 1500

Description When a route is created, it gets its MTU from the egress device. However, if the route is local to the system, there is no egress device. In this case, a default MTU is used.

_ipcfg.SockTtlDefault *Default TTL for Packets Sent via a Socket*

Default Value 64

Description This is the default IP packet TTL value of packets sent via a socket. Note that the application can override this value with the sockets API.

_ipcfg.SockTosDefault *Default TOS for Packets Sent via a Socket*

Default Value 0

Description This is the default IP packet TOS value of packets sent via a socket. Note that the application can override this value with the sockets API.

_ipcfg.SockMaxConnect *Maximum Connections on a Listening Socket*

Default Value 8

Description This is the maximum number of connections a socket will pend waiting for a sockets *accept()* call from the application. Note: This value is also the upper bounds of the maximum connection argument supplied by an application via the sockets *listen()* function (calls with higher values are silently rounded down).

_ipcfg.SockTimeConnect *Maximum Time in Seconds to Wait on a Connect*

Default Value 80

Description This is the maximum amount to time the sockets layer will wait on an actively connecting socket. The default value of 80 is a few seconds longer than the TCP keep time, so TCP will generate the official (more accurate) timeout error.

_ipcfg.SockTimeIo *Maximum Time in Seconds to Wait on Socket Read/Write*

Default Value 0

Description This is the maximum amount of time the sockets layer will wait on a read or write operation without any progress. For example, if the user calls *send()* with a very large buffer, the function will not time out so long as some fraction of the data is sent during the timeout period. After every successful transfer of data, the timeout period is reset. A timeout value of ZERO means never time out.

_ipcfg.SockTcpTxBufSize *TCP Transmit Buffer Size*

Default Value 8192

Description This is the size of the TCP send buffer. A TCP send buffer is allocated for every TCP socket. This value cannot be overridden by the sockets option function.

_ipcfg.SockTcpRxBufSize *TCP Receive Buffer Size (Copy Mode)*

Default Value 8192

Description This is the size of the TCP receive buffer allocated for a standard TCP socket. Note that only SOCK_STREAM sockets use receive buffers. This value cannot be overridden by the sockets option function.

_ipcfg.SockTcpRxLimit *TCP Receive Limit (Non-Copy Mode)*

Default Value 8192

Description This is the maximum number of cumulative bytes contained in packet buffers than can be queued up at any given TCP based socket. Note that only TCP sockets using SOCK_STREAMNC queue packet buffers directly to a socket. This value cannot be overridden by the sockets option function.

_ipcfg.SockUdpRxLimit *UDP Receive Limit*

Default Value 8192

Description This is the maximum number of cumulative bytes contained in packet buffers than can be queued up at any given UDP or RAW based socket. This value cannot be overridden by the sockets option function.

_ipcfg.SockBufMinTx *Min Size in Bytes for Socket "Able to Write"*

Default Value 2048

Description This is the size in bytes required to be free in the TCP buffer before it is regarded as able to write by the system. (Affects how the write fd set behaves in a *select()* call.) This value is usually about 25% to 50% of the send buffer size. UDP and RAW IP sockets are always able to write.

_ipcfg.SockBufMinRx *Min Size in Bytes for Socket "Able to Read"*

Default Value 1

Description This is the size in bytes required to be present in a socket buffer for it to be regarded as able to be read by the system. (Affects how the read fd set behaves in a *select()* call.) Alter at your own risk.

_ipcfg.PipeTimeIo *Maximum Time in Seconds to Wait on Pipe Read/Write*

Default Value 0

Description This is maximum amount to time the file layer will wait on a read or write operation on a pipe without any progress. For example, if the user calls *send()* with a very large buffer, the function will not time out as long as some fraction of the data is sent during the timeout period. After every successful transfer of data, the timeout period is reset. A timeout value of ZERO means never time out.

_ipcfg.PipeBufSize *Size in Bytes of Each End of a Pipe Buffer*

Default Value 1024

Description This is the size of a Pipe send and receive buffer. This value is only examined when pipes are created, so changing this value will not affect the buffering of existing pipes.

_ipcfg.PipeBufMinTx *Min Size in Bytes for Pipe Able to Write*

Default Value 256

Description This is the size in bytes required to be free in the Pipe buffer before it is regarded as able to write by the system. (Affects how the write fd set behaves in a *select()* call.) It is usually about 25% to 50% of the send buffer size. This value is only examined when pipes are created, so changing this value will not affect the buffering of existing pipes.

_ipcfg.PipeBufMinRx *Min Size in Bytes for Pipe "Able to Read"*

Default Value 1

Description This is the size in bytes required to be present in the Pipe receive buffer for it to be regarded as able to be read by the system. (Affects how the read fd set behaves in a *select()* call.) Alter at your own risk. This value is only examined when pipes are created, so changing this value will not affect the buffering of existing pipes.

_ipcfg.TcpKeepIdle *Keep Idle Time (0.1 Sec Units)*

Default Value 72000 (2 hours)

Description This parameter only affects sockets that have specified the SO_KEEPALIVE socket option. It is the time a TCP connection can be idle before KEEP probes are sent.

_ipcfg.TcpKeepIntvl *Keep Probe Interval (0.1 Sec Units)*

Default Value 750 (75 seconds)

Description This parameter only affects sockets that have specified the SO_KEEPALIVE socket option. It specifies the time between probe intervals once TCP begins sending KEEP probes.

_ipcfg.TcpKeepMaxIdlealign *Keep Probe Timeout (0.1 Sec Units)*

Default Value 6000 (10 minutes)

Description This parameter only affects sockets that have specified the SO_KEEPALIVE socket option. It is the time the TCP will continue to send unanswered KEEP probes before timing out the connection.

_ipcfg.IcmpDontReplyBCast *Do NOT Reply to ICMP Echo Request Packets Sent to broadcast/directed broadcast addresses*

Default Value 0 (Reply to ICMP Echo Request packets sent to broadcast/directed broadcast addresses)

Description When set, causes ICMP to not reply to ICMP Echo Request packets sent to broadcast or directed broadcast addresses.

_ipcfg.IcmpDontReplyMCast *Do NOT Reply to ICMP Echo Request Packets Sent to multicast addresses*

Default Value 0 (Reply to ICMP Echo Request packets sent to multicast addresses)

Description When set, causes ICMP to not reply to ICMP Echo Request packets sent to multicast addresses.

A.13 Network Address Translation

A.13.1 Synopsis

The stack includes a small network address translation (NAT) function that can be used to setup virtual networks when the stack is acting as a router. When enabled, NAT will translate routed packets sent from or to a targeted virtual network.

A.13.2 Operation

NAT works by altering the TCP/UDP port numbers of a packet sent from a virtual network, and then using an alternate IP on the physical network to transfer the packet. For ICMP packets, the Id field of ICMP requests is used.

When configured, NAT will have a target virtual network that consists of a IP base address and a subnet mask. It also has a physical IP address that is used as a type of proxy for the translated packets.

The types of packets translated include:

- Any TCP or UDP packet
- ICMP ECHO and TSTAMP packets sent from the virtual network
- ICMP ECHOREPLY and TSTAMPREPLY packets sent to the virtual network
- ICMP error packets sent to the virtual network in response to a translated packet sent from the virtual network

The translation entries are created dynamically, and have a lifetime based on their protocol. ICMP and UDP translation entries have a fixed time limit based on the last time they were accessed. TCP expiration is based on the state of the TCP connection.

Note that some protocols (like FTP) communicate TCP/UDP port information in the packet payload. These types of protocols will not function under NAT without a custom proxy program to alter the packet payload. Individual proxies are not provided.

A.13.3 NAT Configuration

To use NAT, it must be configured via the following function. Also, by default, the NAT code is not called by the stack. This increases stack efficiency when NAT is not in use. To enable the NAT module, the *IpNatEnable* element of the stack configuration structure must be set.

Note that when using the NAT service feature in NETTOOLS or when using the configuration system, this low-level configuration is not required.

NatSetConfig — ***Configure the Network Address Translation Module***

Syntax

void NatSetConfig(IPN IPAddr, IPN IPMask, IPN IPServer, uint MTU);

Parameters

IPAddr	IP address of the Virtual Network
IPMask	IP mask of the Virtual Network
IPSever	Physical IP address of the server that will host the NAT translation
MTU	IP Packet MTU (1500 for Ethernet, 1492 for PPPoE, etc.)

Description

This function configures NAT with a virtual network and a physical server. Note that both the virtual and physical addresses must also be contained in the stack's route table. NAT should only be used when the stack is acting as a router, and when there are more than one Ethernet devices present.

The MTU parameter must be in the range of 64 to 1500. When set less than 1500, TCP connection negotiation will be altered so that TCP sessions through NAT will be limited to the MTU specified. This prevents unnecessary fragmentation when using NAT over dissimilar packet devices. (Note this MTU is the IP packet MTU, not the TCP MTU.)

A.14 Obtaining Stack Statistics

Stack statistics are available from global structures or global arrays exported by the stack library. The declaration of these global identifiers appears in the interface specification for the individual protocols. The following protocols contain statistics information:

Protocol	Statistics Definition
IP	IPIF.H
ICMP	ICMPIF.H
TCP	TCPIF.H
UDP	UDPIF.H
Raw Transport (non-TCP/UDP)	RAWIF.H
Network Address Translation	NATIF.H

Network Address Translation

This section is required only for system programming that needs low level access to the Network Address Translation (NAT) layer. **This API does not apply to sockets application programming.**

This section describes functions that are callable from the kernel layer. You should be familiar with the operation of the operation of *llEnter()llExit()* functions before attempting to use the APIs described in this section (see Section A.1.2).

NAT has a unique status in the stack software because it can be an integral part of programming at both the user and kernel levels, or can be entirely redundant and even purged from the stack build.

This section describes the operation of the Network Address Translation software included in the NDK, how to configure it, how to install port mappings, and how to program proxy filter routines to support protocols like FTP.

Topic **Page**

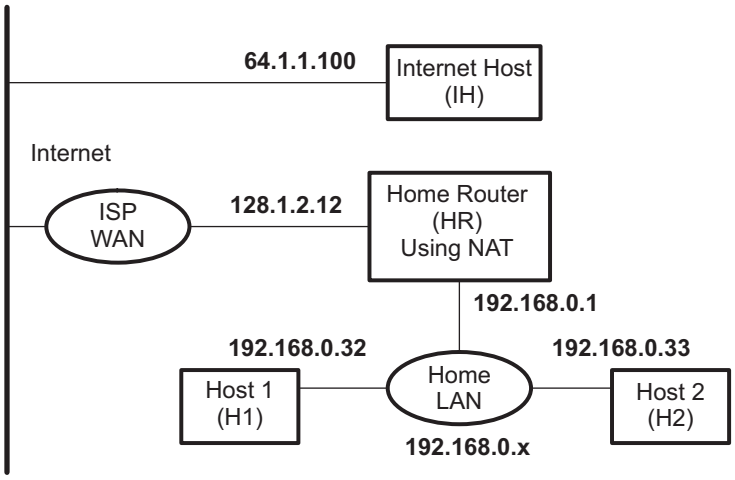
B.1 NAT Operation

NAT is a translation of packet IP address. It is used by the stack when routing, to translate the IP address of a packet to/from a private LAN from/to a public WAN. NAT is required when the IP address paradigms on either side of the router are incompatible; for example, virtual addresses vs. physical addresses, or private vs. public. In the case of a home LAN, NAT allows multiple clients on the home LAN to use a single ISP account by sharing the router WAN IP address obtained from the ISP.

B.1.1 Typical Configuration

For the examples that follow, consider the typical configuration illustrated in Figure B-1. The NDK is executing as a home router (HR) and connects the home LAN subnet (192.168.0.x) to the Internet (WAN) via an ISP that has assigned HR an address of 128.1.2.12. The hosts on the home network (H1 and H2) have obtained their internet addresses from HR via DHCP. The IP of HR on the home LAN as well as the IP subnet used by the home LAN is pre-configured in HR. Figure B-1 also shows a host on the public internet (IH) to which the LAN hosts will connect. Lastly, it is assumed that the home LAN subnet is virtual, and NAT is required to allow H1 and H2 to share the WAN IP address assigned to HR by the ISP (128.1.2.12).

Figure B-1. Basic Home Network Configuration



B.1.2 Basic NAT

When sharing a single WAN IP address, the IP address obtained from the ISP is assigned to the router (the NDK in routing mode). Client machines that are to share the IP address are placed on the home LAN. The router routes traffic between the LAN and the WAN (internet via the ISP).

As packets traverse from the LAN to the WAN across the router, the source IP address of the packet (a LAN address) is replaced with the public IP address of the router. The result is that all packets sent to the WAN appear to have originated from the router with the public IP address obtained from the ISP.

As packets traverse from the WAN to the LAN across the router, the destination IP address of the packet (the router's WAN IP as obtained from the ISP) is replaced with the home LAN IP address of the physical client machine to which the packet is ultimately destined.

To perform this translation successfully, some details must be addressed. First, to allow multiple clients to share the public IP address in a non-ambiguous fashion, there must exist a deterministic method of mapping packets from the WAN to their correct destination on the LAN. This is done by keeping records of LAN IP clients that have initiated IP traffic, and by altering the TCP/UDP port (or ICMP Id field) as well as the IP address when performing the translation.

Every time a LAN client sends a packet to the WAN, the local IP address, port/id, and protocol is recorded for reverse mapping, as well as the destination IP address and port for security. When a packet is received from the WAN, the destination port/id is checked against the current database of NAT entries to see if the packet's destination address and port/id should be translated to a LAN client.

For example, when accessing the Internet, all communication is normally initiated by the client. In this case, communication will be initiated by H1 or H2. Assume that H1 attempts to establish an HTTP connection with the Internet host (IH). It will send a connection request to the IP address assigned to IH, and a TCP port value of 80, which is HTTP. The request will be from its own IP address with an ephemeral port value that is picked from a pool (consider it random for these purposes- for example, 1001). So the request will be addressed as follows:

Packet 1

To	From	Protocol
64.1.1.100 : 80	192.168.0.32 : 1001	TCP

When the router HR receives this packet, it searches for a NAT entry that matches the *From* address of the packet. Because this is the first packet, assume the table is empty. When no entry is found, (skipping proxies for now) the router will create a new entry. It does this by recording information from packet 1, as well as picking a new port value from its own pool that has been specifically reserved for NAT (assume the range is 50000 to 55000, and that it chooses 50001). The new port is used as the packet's source port. The NAT entry record would look like the following:

NAT Entry Table

Foreign IP	Foreign Port	Local IP	Local Port	Mapped Port	IP Protocol	TCP State	Timeout
64.1.1.100	80	192.168.0.32	1001	50001	TCP	SYNSENT	00:01:00

The *Local IP* and *Local Port* values are those that are local to hosts on the home LAN. The *Foreign IP* value is the foreign side of the connection as viewed by hosts on the home LAN. The *Mapped Port* value is the source port when the packet is sent from HR. The source IP address used in the packet is that assigned to HR by the ISP. The IP protocol of the packet is recorded, and when using TCP, the state of the TCP connection is tracked to establish a reasonable timeout value. The SYNSENT value indicates that a connection request was sent. Before a full connection is established, the timeout is set fairly low - for example, 1 minute.

As the packet is transmitted from HR to the ISP, it would look like the following:

Packet 1 (modified)

To	From	Protocol
64.1.1.100 : 80	128.1.2.12 : 50001	TCP

When IH receives the packet, it believes that the connection request came from HR. It thus sends the response packet to HR. The packet would be addressed as follows:

Packet 2 (response to packet 1)

To	From	Protocol
128.1.2.12 : 50001	64.1.1.100 : 80	TCP

When HR receives the packet, it checks the NAT entry table for an entry with a *Mapped Port* value equal to the destination port of the packet (in this case 50001). The value of *Protocol* and the source IP address/port values must also match the *Protocol*, *Foreign IP*, and *Foreign Port* fields of the NAT entry. This helps ensure that the reply is from the desired server.

Here, HR finds the entry and proceeds to modify the packet. It replaces the destination address/port with the local address/port stored in the entry. It also resets the timeout of the entry. After modification, the packet would be addressed as follows:

Packet 2 (modified)

To	From	Protocol
192.168.0.32 : 1001	64.1.1.100 : 80	TCP

Once a connection is established, the timeout of the entry is set high (for example, five hours), because TCP connections can stay connected for an indefinite period of time without exchanging any packets.

If H2 attempts to connect to the same host simultaneously, it can share the public IP address assigned to HR. For example, H2 sends a connection request to IH addressed as follows:

Packet 3

To	From	Protocol
64.1.1.100 : 80	192.168.0.33 : 1024	TCP

HR would not find a NAT entry for 192.168.0.33:1024, so it would create one:

NAT Entry Table

Foreign IP	Foreign Port	Local IP	Local Port	Mapped Port	IP Protocol	TCP State	Timeout
64.1.1.100	80	192.168.0.33	1024	50002	TCP	SYNSENT	00:01:00
64.1.1.100	80	192.168.0.32	1001	50001	TCP	CONNECT	04:59:23

The modified packet and its reply would proceed similar to packets 1 and 2. Packets that pass from the LAN to the WAN are searched based on *Local IP* combined with *Local Port*. Packets that pass from the WAN to the LAN are searched based on *Mapped Port*. Note that for all entries on the NAT entry table, these values are unique.

B.1.3 NAT Port Mapping

So far, you have examined communication that has been initiated by hosts on the home LAN. Note that any unsolicited packets sent to HR from the WAN will not match any entry in the NAT table. These packets will be forwarded to the internal protocol stacks on HR, where they may or may not be used.

Now assume that a host on the home LAN (for example, H2) must place an HTTP server on the Internet. With what has been examined so far, it would be impossible to contact such a server from the WAN because no unsolicited traffic (like an HTTP connect request) can pass from the WAN to the LAN. However, H2 can acquire a portion of HR's WAN presence by mapping one of the well-known port values on the public WAN IP address to itself through port mapping.

In port mapping, a NAT entry is created to send all traffic destined for a specific port on the public IP address to an alternate destination. For well known ports like HTTP, the port value is not usually altered. Only the destination IP address changes. In this case, port 80 (HTTP) on the public IP address is mapped to port 80 of the LAN host H2. The entry would look as follows:

NAT Entry Table

Foreign IP	Foreign Port	Local IP	Local Port	Mapped Port	IP Protocol	TCP State	Timeout
wild	*wild*	192.168.0.32	80	80	TCP	–	STATIC

When a connection request arrives from a remote host for the public IP address assigned to HR, as with the basic NAT discussion of the previous section, the destination port of the packet is matched with the *Mapped Port* value of the NAT entry. Normally, the *Foreign IP* and *Port* of the NAT entry must also match for source IP and port of the packet, but here the values are *wild*. This is because when the entry is created, the foreign peer is unknown. Because, every TCP connection state must be tracked in its own NAT entry, a second entry must be spawned. Any match of a wild NAT entry will spawn a fully qualified entry. For example, assume the following packet arrives:

Packet 4

To	From	Protocol
128.1.2.12 : 80	64.1.1.100 : 2006	TCP

The resulting NAT entry table would be:

NAT Entry Table

Foreign IP	Foreign Port	Local IP	Local Port	Mapped Port	IP Protocol	TCP State	Timeout
64.1.1.100	2006	192.168.0.32	80	80	TCP	SYNSENT	00:01:00
wild	*wild*	192.168.0.32	80	80	TCP	–	STATIC

The packet sent to the LAN by HR would be:

Packet 4 (modified)

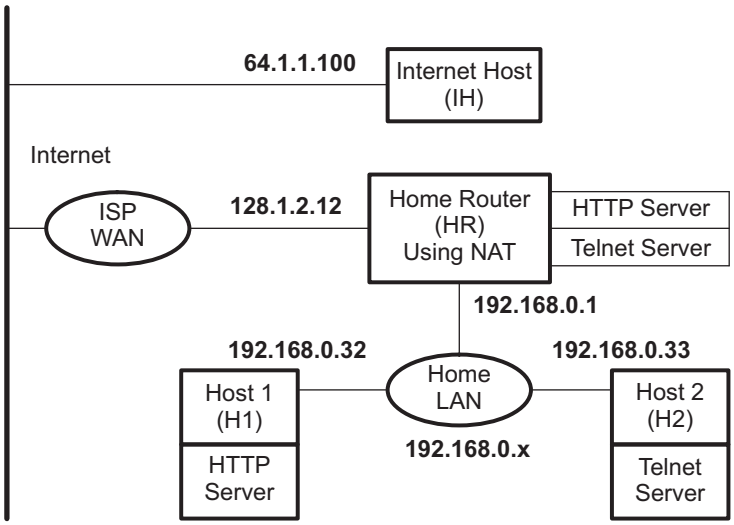
To	From	Protocol
192.168.0.32 : 80	64.1.1.100 : 2006	TCP

Note that the wildcard entry's timeout is STATIC. This means that the entry will never expire. Note that when the new entry is spawned, it acquires a timeout.

One last point to note here is that the installation of a port map for port 80 does not prohibit HR from running its own HTTP server hosted on its private LAN IP address (192.168.0.1). This means that local hosts could get to a local HTTP server on 192.168.0.1, and the public HTTP server on 192.168.0.32, but outside hosts connecting to 128.1.2.12 could only get to the public HTTP server on 192.168.0.32.

For example, assume the same topology as before, with the HR running both and HTTP and Telnet servers, H1 running an HTTP server, and H2 running a Telnet server. This is illustrated in Figure B-2.

Figure B-2. Public Servers on the Home Network



To make the servers on H1 and H2 public, the following NAT port mapping entries are installed on HR:

NAT Entry Table

Foreign IP	Foreign Port	Local IP	Local Port	Mapped Port	IP Protocol	TCP State	Timeout
wild	*wild*	192.168.0.33	23	23	TCP	–	STATIC
wild	*wild*	192.168.0.32	90	90	TCP	–	STATIC

With these mappings, the externally available HTTP server and Telnet server publicly accessible on the WAN IP (128.1.2.12) are actually executing on H1 and H2. However, HR can have its own HTTP and Telnet servers and make them available to hosts on the LAN.

Also note that, regardless of how hosts on the LAN access HR (either through 192.168.0.1 or 128.1.2.12), their packets are not processed via NAT. Thus, they are never altered. The following are some connection examples:

Client	Protocol Used	Target Address	Resulting Server Connection
IH	HTTP	128.1.2.12	HTTP on H1
H2	HTTP	128.1.2.12	HTTP on HR
H2	HTTP	192.168.0.1	HTTP on HR
H2	HTTP	192.168.0.32	HTTP on H1
IH	Telnet	128.1.2.12	Telnet on H2
H1	Telnet	128.1.2.12	Telnet on HR
H1	Telnet	192.168.0.1	Telnet on HR
H1	Telnet	192.168.0.33	Telnet on H2

B.1.4 NAT Proxy Filters

B.1.4.1 Problem Synopsis

Translating the IP destination address of a packet via NAT guarantees that all packets can be redirected to their correct physical destination, but it does not guarantee that the information will be understood by the recipient. Because one side of the connection always believes they are actually connected to a different IP address than their physical peer, there is a possibility that the application using the information will become confused. The confusion arises when there is information in the packet payload that is dependent on the IP address/port of the peer connection.

B.1.4.2 Problem Example - FTP Clients on the LAN

As a straightforward example of a situation that requires a proxy filter, consider FTP (file transfer protocol). FTP actually uses two ports to transmit a file. The first port establishes the control connection. Then, new ports establish the data connection to actually send the file. The FTP protocol allows an FTP client to specify its port for the data connection to the server. If no port is specified by the client, the client's control port value is used.

The above scenario presents a couple problems for standard NAT. First, if NAT creates an entry for the FTP control connection, the entry could not be used for the data connection. As an example, H1 sends an FTP connection request to IH. The packet would be addressed as follows:

Packet 1

To	From	Protocol
64.1.1.100 : 21	192.168.0.32 : 1137	TCP

HR would not find a NAT entry for 192.168.0.33:1137, so it would create one:

NAT Entry Table

Foreign IP	Foreign Port	Local IP	Local Port	Mapped Port	IP Protocol	TCP State	Timeout
64.1.1.100	21	192.168.0.32	1137	50003	TCP	SYNSENT	00:01:00

The modified packet and its reply would proceed as discussed in Section B.1.2. The modified packet would be:

Packet 1 (modified)

To	From	Protocol
64.1.1.100 : 21	128.1.2.12 : 50003	TCP

Now assume that eventually the FTP server on IH attempts to establish a data connection back to what it thinks is the FTP client's ephemeral port (50003). Note classic FTP uses port 20 to establish data connections. Its connection request packet would be:

Packet 2 (Data connection request)

To	From	Protocol
128.1.2.12 : 50003	64.1.1.100 : 20	TCP

Because there is no NAT entry record that will match the address values in this packet (specifically port 20 in the *From* field), this packet will not be forwarded to the FTP client. For this to work, there must be a port mapping installed for 64.1.1.100 that has a wildcard port value (it is not certain that the connection request will arrive on port 20). The NAT entry table would be as follows:

NAT Entry Table

Foreign IP	Foreign Port	Local IP	Local Port	Mapped Port	IP Protocol	TCP State	Timeout
64.1.1.100	*wild*	192.168.0.32	1137	50003	TCP	–	STATIC
64.1.1.100	21	192.168.0.32	1137	50003	TCP	CONNECT	04:58:39

With such a mapping, if a connection request from port 20 arrived, the wild card entry would be matched, and another entry spawned for port 20 on IH. The table would look as follows:

NAT Entry Table

Foreign IP	Foreign Port	Local IP	Local Port	Mapped Port	IP Protocol	TCP State	Timeout
64.1.1.100	20	192.168.0.32	1137	50003	TCP	SYNSENT	00:01:00
64.1.1.100	*wild*	192.168.0.32	1137	50003	TCP	–	STATIC
64.1.1.100	21	192.168.0.32	1137	50003	TCP	CONNECT	04:58:39

The second issue in dealing with an FTP client is that the client can change the port on which the FTP server attempts connection. This is done via a PORT command sent from the client to the server. The PORT command contains information about the client in the packet payload.

For example, assume the FTP client (H1) creates a new socket for the data connection, and its ephemeral port value is 1142. H1 would then send an FTP PORT command on the control connection to the server. The server would then attempt a connection. The following is an approximation of the operation (it is not the exact syntax of the port command).

Packet 3 (FTP Client H1 Sends Port Command for Port 1142)

To	From	Protocol	Packet Payload
64.1.1.100 : 21	192.168.0.32 : 1137	TCP	"PORT 192.168.0.32, 1142"

As a reminder, the FTP server would normally see the packet as:

Packet 3 (modified incorrectly)

To	From	Protocol	Packet Payload
64.1.1.100 : 21	128.1.2.12 : 50003	TCP	"PORT 192.168.0.32, 1142"

This packet creates a couple of problems. First, the IP address in the PORT command does not match the IP address of the FTP server's connected peer. This would produce an error. Plus, the IP address in the PORT command is not a real Internet address. Lastly, even if the FTP server tried to connect to 128.1.2.12:1142, there is no mapping for the port number in the NAT entry table.

The correct procedure for modifying this packet is to solve all the above problems. First, a new NAT entry is created for 192.168.0.32:1142. The foreign IP address is left as a wildcard because as before, because it is not certain what port the FTP server will use. The NAT entry table would then look as follows:

NAT Entry Table

Foreign IP	Foreign Port	Local IP	Local Port	Mapped Port	IP Protocol	TCP State	Timeout
64.1.1.100	*wild*	192.168.0.32	1142	50004	TCP	–	00:02:00
64.1.1.100	*wild*	192.168.0.32	1137	50003	TCP	–	STATIC
64.1.1.100	21	192.168.0.32	1137	50003	TCP	CONNECT	04:58:39

To review, note that you have the original NAT entry for the FTP control connection, and now two wildcard entries for possible FTP data connection requests.

The final step of the modification is to alter the payload of the packet so that the information in the PORT command matches the WAN IP address of HR (128.1.1.21) and the *Mapped Port* of the new NAT entry (50004). The correctly modified packet would be:

Packet 3 (modified correctly)

To	From	Protocol	Packet Payload
64.1.1.100 : 21	128.1.2.12 : 50003	TCP	"PORT 128.1.2.12, 50004"

It is also possible for a client to request the FTP server to create a new port (the PASV command), but that does not create any issues for FTP clients on the LAN. If the FTP server were on the LAN and the client on the WAN, the proxy process would key off the PASV command.

B.1.4.3 NDK Support for Proxy Filters

The modification procedure discussed above does have some multifaceted problems:

1. The creation of the first data connection wildcard entry depends on the knowledge by some entity that an FTP control connection has occurred, and what IP/PORT the connection occurred on.
2. The creation of the second data connection wildcard entry depends on the detection of a PORT command being passed from the client to the server.
3. The modification of the data payload of the packet containing the PORT command requires that some entity is examining packet payloads.
4. Modification of a TCP packet payload can permanently alter the values of the TCP sequence and acknowledge fields in the TCP header of all future packets on the control connection.

The first three problems are very specific to FTP, and the fourth problem (TCP sequence) is specific to any alteration of a TCP packet payload. Fortunately, the proxy filter support routines remove much of the burden of supporting these transformations.

The solution is twofold. First, the stack allows you to install proxy filter callback functions on specified TCP/UDP port values, either outgoing (for clients) or incoming (for servers). There are three callback functions involved.

The first callback function *Enable* is called when a new connection is attempted, or when the NAT entry expires. This function allows you to establish the basic connection state for the protocol in question. In the case of the FTP client example, the first wildcard data connection mapping would be installed here. Note that this function can also be used to filter connection requests. If this function returns zero, the connection request is ignored.

The second and third callback functions are mirrors of the other. They are the Tx and Rx functions. The Tx callback is called with the IP header of every packet that passes from the LAN to the WAN for the connection in question, while the Rx callback is called with the IP header of every packet that passes from the WAN to the LAN. While in these functions, the programmer can call a packet modify function to modify the payload of the packet. The system will automatically track and perform modifications to the TCP sequence values (when using TCP).

In the case of the FTP client, there would be no Rx callback because only packets from the client need to be examined. The Tx callback would look for PORT commands from the client, and when one was detected, it would install the second wildcard port mapping as discussed in the previous section, and then modify the packet payload so that the PORT command reflected the WAN IP of HR, and the Mapped Port of the NAT entry.

B.1.4.4 FTP Proxy Filter Example Code

From the discussion in this section, it would be easy to draw the conclusion that developing proxy filter code would be horribly complicated. However, the actual implementation is straightforward. The code to implement the filter discussed in Section B.1.4.3 is shown below. The API for NAT and Proxy is discussed in the following sections.

```
//
// GetVal - Convert ASCII decimal string to integer
//
static uint GetVal( UINT8 **pData )
{
    uint v = 0;
    while(**pData >= '0' && **pData <= '9')
        v = v*10 + (*(*pData)++ - '0');
    (*pData)++;
    return(v);
}

//
// FTPCProxyEnable - Proxy for FTP Clients behind firewall
//
// NOTE: Proxy callback function operate at the kernel level. They
//       may not make calls to user-level functions.
//
int FTPCProxyEnable( NATINFO *pin, uint Enable )
{
    HANDLE hNat;

    // Some implementations of FTP require the host to listen for
    // connections on the ephemeral port used to connect to the FTP
    // server. We create a STATIC mapping to handle this.
    if( Enable )
    {
        // Create it
        hNat = NatNew( pNI->IPLocal, pNI->PortLocal, pNI->IPForeign, 0,
                       IPPROTO_TCP, pNI->PortMapped, 0 );
        pNI->pUserData = hNat;
    }
    else
    {
        // Destroy it
        NatFree( pNI->pUserData );
    }
    return(1);
}

//
// FTPCProxyTx - Proxy for FTP Clients behind firewall
//
// NOTE: Proxy callback function operate at the kernel level. They
//       may not make calls to user-level functions.
//
int FTPCProxyTx( NATINFO *pNI, IPHDR *pIpHdr )
{
    UINT16  Length, Offset;
    TCPHDR  *pTcpHdr;
    UINT8   *pData;
    HANDLE  hNAT;
    NATINFO *pNINew;
    char    tmpstr[32];
    UINT16  PortNew;
    IPN     IPNew;

    pData = (UINT8*)pIpHdr;
    // Get pointer to TCP header
    Offset  = (pIpHdr->VerLen & 0xf) * 4;
```

```
        pTcpHdr = (TCPHDR *)(pData + Offset);

        // Get length of the IP payload
        Length = HNC16(pIpHdr->TotalLen) - Offset;

        // Get the offset into the TCP payload and payload size
        Offset += pTcpHdr->HdrLen >> 2;
        Length -= pTcpHdr->HdrLen >> 2;

        // Get pointer to TCP payload
        pData  += Offset;

        //
        // For clients, we only care about sending PORT commands
        //
        // For example, if our client IP is 192.138.139.32, and it reports
        // port 384, the form of the command sent to the FTP server would
        // be: "PORT 192,138,139,32,1,128\r\n"
        //
        // We replace the Client IP with the router's IP, and the client
        // port with a NAT port which is mapped to the client port.
        //
        if(!strncmp( pData, "PORT ", 5) )
        {
            // Get the IP/Port declared by sender
            // Form is "i1,i2,i3,i4,p1,p2"
            pData += 5;
            IPNew   = ((UINT32)GetVal (&pDada)) << 24;
            IPNew  |= ((UINT32)GetVal (&pDada)) << 16;
            IPNew  |= ((UINT32)GetVal (&pDada)) << 8;
            IPNew  |= ((UINT32)GetVal (&pData));
            IPNew = htonl(IPNew);
            PortNew = GetVal(&pData);
            PortNew = (PortNew<<8) + GetVal (&pData);

            // Add a NAT mapping to client's IP and Port
            hNAT = NatNew(IPNew, PortNew, pNI->IPForeign, 0, IPPROTO_TCP,
                          0, NAT_IDLE_SECONDS);
            if(!hNAT)
                return(0);

            // Get Server IP and Mapped Port
            IPNew   = htonl( NatIpServer );
            pNINew  = NatGetPNI( hNAT );
            PortNew = pNINew->PortMapped;

            // Print a replacement string with IP and Port
            sprintf(tmpstr, "%u,%u,%u,%u,%u,%u\r\n",
                             ((uint)(IPNew >> 24)),
                             ((uint)(IPNew >> 16)&0xFF),
                             ((uint)(IPNew >> 8)&0xFF),
                             ((uint)(IPNew)&0xFF),
                             PortNew>>8, PortNew&0xFF);

            // Replace the original string with ours
            ProxyPacketMod( Offset+5, Length-5, strlen(tmpstr), tmpstr );
        }

        return(1);
    }
```

B.2 NAT Port Mapping

B.2.1 Synopsis

NAT port mapping allows a client machine on the LAN (or home network) to appear on a specific port of the router's public WAN IP address. This API (and NAT in general) is only used when the NDK is acting as an IP router, and when the IP network on one side of the router is using virtual IP addresses.

The functions described in this section illustrates how to install and remove port mappings. The functional operation of NAT and NAT Port Mapping is discussed in more detail in Section B.1.

B.2.2 Function Overview

The following functions create and destroy port mappings:

NatNew()	Create a new NAT entry (for port mapping)
NatFree()	Free a NAT entry
NatGetPNI()	Get a pointer to a NAT entry's NATINFO structure

B.2.3 NAT Entry Information Structure

A port mapping is just a NAT entry. Each NAT entry has its own information structure. This NATINFO structure allows you to examine the status of a particular entry.

The specification of the NATINFO structure is as follows:

```
typedef struct _natinfo {
    uint                TcpState;    // Current TCP State (Simplified)
#define NI_TCP_CLOSED       0            // Closed or closing
#define NI_TCP_SYNSENT      1            // Connecting
#define NI_TCP_ESTAB        2            // Established
    IPN                 IPLocal;     // Translated IP Address
    UINT16              PortLocal;   // Translated TCP/UDP Port
    IPN                 IPForeign;   // IP Address of Foreign Peer
    UINT16              PortForeign; // Port of Foreign Peer
    UINT8               Protocol;    // IP Potocol
    UINT16              PortMapped;  // Locally Mapped TCP/UDP Port (router)
    HANDLE              hProxyEntry; // Handle to Proxy Entry (if any)
    UINT32              Timeout;     // Expiration time in SECONDS
    void                *pUserData;  // Pointer to proxy callback data
    } NATINFO;
```

The individual fields are defined as follows:

- `uint TcpState;`

  This is a condensed version of the state of the TCP connection that is being translated by this entry. This field is only valid when the Protocol field is set to IPPROTO_TCP. The defined values are:

NI_TCP_CLOSED	The connection is closed
NI_TCP_SYNSENT	The peers are in the process of connecting
NI_TCP_ESTAB	A connection has been established

- `IPN IPLocal;`

  This is the IP address (in network format) of the peer host on the local network (LAN). It is the entity that has been assigned a virtual IP address behind the firewall.

- `UINT16 PortLocal;`

  This is the port in use by the peer host on the local network (LAN). It is the entity that has been assigned a virtual IP address behind the firewall.

- `IPN IPForeign;`
  This is the IP address (in network format) of the peer host on the public network (WAN). It is the entity that is on the physical network outside the firewall.
- `UINT16 PortForeign;`
  This is the port in use by the peer host on the public network (WAN). It is the entity that is on the physical network outside the firewall.
- `UINT8 Protocol;`
  This is protocol in use by the NAT entry. It must be IPPROTO_TCP, IPPROTO_UDP, or IPPROTO_ICMP.
- `UINT16 PortMapped;`
  This is the port in use by the router on its public (WAN) IP address. It is this port that maps back to a specific local IP/port on the LAN.
- `HANDLE hProxyEntry;`
  When a NAT entry is created as a result of a proxy filter being installed on a specific port, the HANDLE to the proxy filter that spawned the NAT entry is stored here.
- `UINT32 Timeout;`
  This is time in seconds when the proxy entry will expire. The system checks with a fairly large granularity, so the actual expiration can occur 10 to 20 seconds later. If this value is ZERO, the entry is static. A NAT entry must be specified as STATIC when it is created. Setting Timeout to ZERO will cause the entry to expire in 0 to 20 seconds.
- `void * pUserData;`
  This field is reserved for use by proxy filter callback functions. It is not used by the system software.

The NAT information structure is of little importance when only port mapping is required. It is mostly for use in NAT proxy filters.

B.2.4 NAT API Functions

NatNew — ***Create a NAT Entry (for Port Mapping)***

Syntax

HANDLE NatNew(IPN IPLocal, UINT16 PortLocal, IPN IPForeign, UINT16 PortForeign, UINT8 Protocol, UINT16 PortMapped, UINT32 Timeout);

Parameters

IPLocal	IP address (in network format) of host on the LAN to map
PortLocal	TCP/UDP port value of host on the LAN to map
IPForeign	IP address of WAN peer (usually NULL/wild for port mappings)
PortForeign	TCP/UDP port of WAN peer (usually NULL/wild)
Protocol	IP protocol (IPPROTO_TCP or IPPROTO_UDP)
PortMapped	Port on router's public WAN to map (usually a well-known port)
Timeout	Timeout of entry in seconds (NULL for STATIC)

Return Value

Handle to NAT entry, or NULL on error.

Description

This function creates a NAT entry with the parameters as specified.

For example, to allow a host on a virtual IP address of 1.2.3.4 to run a Telnet server reachable via the router's public (physical) IP address, a mapping would be installed to map TCP port 23 (telnet) to 1.2.3.4:23. If the connection were to be open to all foreign hosts, then *IPForeign* and *PortForeign* would be left NULL. The value of *Timeout* would also be NULL to make the entry STATIC.

```
hNatTelnet = NatNew( htonl(0x01020304), 23, 0, 0, IPPROTO_TCP, 23, 0 );
```

The function returns a handle to the NAT entry created. This handle should be freed with

NatFree() when the mapping is no longer desired.

NatFree — *Destroy a NAT Entry*

Syntax

void NatFree(HANDLE hNat);

Parameters

hNat	Handle to NAT entry created with *NatNew()*

Return Value

None.

Description

This function frees the supplied NAT entry. It is called to remove a STATIC NAT entry that is no longer required.

NatGetPNI — *Get a Pointer to a NAT Entry's NATINFO Structure*

Syntax

NATINFO NatGetPNI(HANDLE hNat);

Parameters

hNat	Handle to NAT entry created with *NatNew()*

Return Value

Pointer to NATINFO structure or NULL on error.

Description

This function returns a pointer to the NATINFO structure defined in Section B.2.3. It is used mainly by NAT proxy filter callback functions.

B.3 NAT Proxy Filters

B.3.1 Synopsis

NAT proxy filters allow NAT to operate correctly with network protocols that have addressing specific data in their packet payload data. This API (and NAT in general) is only used when the NDK is acting as an IP router, and when the IP network on one side of the router is using virtual IP addresses.

The functions described in this section illustrate how to install and remove port proxy filters and their associated callback functions. The functional operation of NAT and NAT Port Mapping, and NAT Proxy is discussed in more detail in Section B.2.3.

B.3.2 Function Overview

The following functions create and destroy proxy filters:

ProxyNew()	Create Proxy Filter for NAT entries
ProxyFree()	Destroy a Proxy Filter declaration

The following function can be called from within a proxy filter callback function:

ProxyPacketMod()	Modify a packet being processed by NAT

B.3.3 NAT Proxy Filter Callback Functions

The proxy filter callback functions allow the proxy programmer to examine NAT entry properties as the entries are created, plus the examination of packet data as packets pass between the LAN and WAN. This section describes the syntax of the callback functions that are supplied to the proxy filter when it is first installed in the system.

ProxyEnableCallback *Proxy Enable Callback Function*

Syntax

int SampleProxyEnableCallback(NATINFO *pNI, uint EnableFlag);

Parameters

pNI	Pointer to NATINFO structure of NAT entry created
EnableFlag	Set to 1 for an enable request

Return Value

1 to allow normal operation, or NULL to abort new NAT entry.

Description

This function is called when a NAT entry containing a proxy is created or destroyed. When the entry is created, the value of *EnableFlag* is 1. When the entry is being destroyed, the value of *EnableFlag* is zero.

When *EnableFlag* is set, the return value of this function determines if the NAT entry will be enabled. If this function returns NULL, the NAT entry is immediately destroyed (in this event, the callback is not called a second time for this destroy). This can be used to restrict peer connections.

ProxyTx/RxCallback *Proxy Tx/Rx Callback Functions*

Syntax

int SampleProxyTxCallback(NATINFO *pNI, IPHDR *pIpHdr);

int SampleProxyRxCallback(NATINFO *pNI, IPHDR *pIpHdr);

Parameters

pNI	Pointer to NATINFO structure of NAT entry in use
pIpHdr	Pointer to the IP header of the packet being translated

Return Value

1 to allow normal operation, or NULL to abort the supplied packet.

Description

This function is called when a packet is crossing the router from the WAN to the LAN (Rx callback) or from the LAN to the WAN (Tx callback). The NAT entry containing a proxy that matches the packet is described by the supplied NATINFO structure. This structure was described in Section B.2.3.

The purpose of the callback is to examine the packet and take appropriate action based on its contents. The packet payload can be easily modified by the *ProxyPacketMod()* function described later in this section. The translation of the IP address and port information cannot be altered by this callback; however, the callback can act as a packet filter and discard unwanted packets by returning a value of NULL.

B.3.4 NAT Proxy API Functions

ProxyNew — *Create a New Proxy Filter for NAT Entries*

Syntax

HANDLE ProxyNew(uint NatMode, UINT8 Protocol, UINT16 Port, IPN IPTarget, int (*pfnEnableCb)(NATINFO *, uint), int (*pfnTxCb)(NATINFO *, IPHDR *), int (*pfnRxCb)(NATINFO *, IPHDR *));

Parameters

NatMode	Port direction to detect (NAT_MODE_RX or NAT_MODE_TX)
Protocol	Protocol to use (IPPROTO_TCP or IPPROTO_UDP)
Port	Port value for RX or TX packets to detect
IPTarget	New IP destination NAT_MODE_RX proxy
pfnEnableCb	Pointer to proxy *Enable* callback function (NULL if none)
pfnTxCb	Pointer to proxy Tx callback function (NULL if none)
pfnRxCb	Pointer to proxy Rx callback function (NULL if none)

Return Value

Handle to new proxy, or NULL on error.

Description

This function creates a hook that is examined whenever a new NAT entry is created.

The calling parameter *NatMode* specifies the direction of the proxy (NAT_MODE_RX for servers behind the firewall, and NAT_MODE_TX for clients behind the firewall).

The *Protocol* and *Port* values are the IP protocol and well known port of the protocol to proxy.

For example, if setting up a FTP client proxy, set:

NatMode = NAT_MODE_TX, *Protocol* = IPPROTO_TCP, and *Port* = 21.

IPTarget is used only in server proxies (when *NatMode* is set to NAT_MODE_RX). This specifies the machine behind the firewall that is actually providing the service.

The three pointers to callback functions correspond to the proxy filter callback functions described in the previous section.

The function returns a handle to the new proxy. Note that a proxy handle is not the same as (or compatible with) a NAT entry handle.

The proxy should be destroyed by calling *ProxyFree()* when it is no longer needed.

ProxyFree — *Destroy a Proxy Filter Declaration*

Syntax

void ProxyFree(HANDLE hProxy);

Parameters

hProxy	Handle to Proxy Filter entry created with *ProxyNew()*

Return Value

None.

Description

This function frees the supplied Proxy Filter entry. It is called to remove an entry that is no longer required.

ProxyPacketMod *Modify the Contents of a Packet*

Syntax

IPHDR *ProxyPacketMod(uint Offset, uint OldSize, uint NewSize, UINT8 *pNewData);

Parameters

Offset	Offset in bytes from start of IP header to first modified byte
OldSize	Size of old data at *Offset*
NewSize	Size of new data to replace old data at *Offset*
pNewData	Pointer to new data to replace old data

Return Value

Pointer to new IP header of packet. This pointer is used for further modifications (if needed).

Description

This function may only be called from a proxy filter callback function. Its purpose is to modify the contents of a TCP or UDP packet, and perform the necessary adjustments for packet size - including TCP sequencing adjustment.

Point-to-Point Protocol

Point to point protocol (PPP) was originally designed as a replacement for SLIP (serial line IP) in sending IP packets via a serial line. In addition to its massive popularity in performing this function, PPP has also been increasingly used for the transmission of packets over other media. This is due to PPP's inherent peer-to-peer nature, allowing for per-connection security and billing.

The NDK has built-in support for both PPP servers and clients. The PPP support API is designed to be shared by one or more physical devices. One obvious device that can be hooked to PPP is a serial line, but the stack also contains support for PPP over Ethernet (PPPoE). The low level PPP API as well as Serial HDLC and PPPoE are all discussed in this appendix.

Topic **Page**

C.1 Low Level PPP Support

This section describes the operation of the PPP support API included in the NDK.

Note: Unlike the HDLC and PPPoE APIs that are application callable, the low level PPP support API is designed to be called from the kernel layer only. You should be thoroughly familiar with the operation of the kernel and the *llEnter()/llExit()* functions before attempting to use the APIs described in this section.

C.1.1 PPP Operation

PPP is very much like Ethernet in that there is a defined packet format. The basic PPP packet is shown below. It consists of flag delimiters, address and control bytes, protocol field (similar to ether-type under Ethernet), and a two byte checksum.

Figure C-1. Standard PPP Frame Over Serial Line

Flag (7E)	Addr (FF)	Control (03)	Protocol	Payload	CRC	Flag (7E)
1	1	1	2	1500	2	1

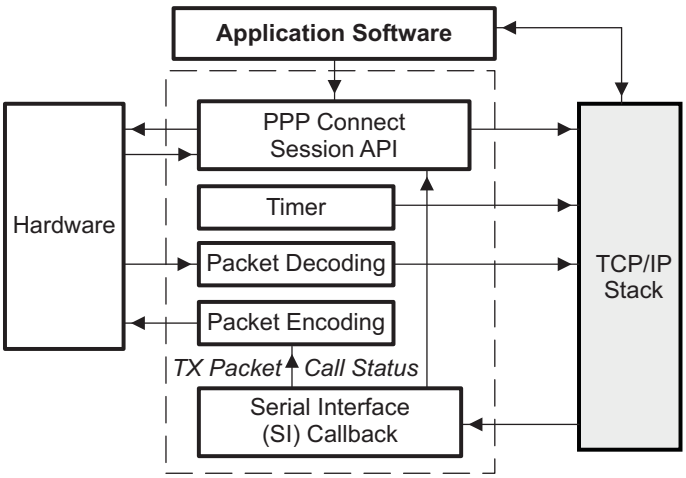
To abstract out the actual processing of the PPP data from the processing of the PPP frame encoding, the PPP support included in the NDK expects a smaller frame, consisting of the protocol and payload fields only. This format is shown in Figure C-2.

Figure C-2. PPP Frame Processed by PPP API

Protocol	Payload
2	Size specified by layer 2 (about 1500)

The abstraction of PPP from the layer 2 encoding allows PPP to be carried by a variety of physical devices. The programming interface to the PPP layer called by the application is actually exposed by the layer 2 encoder. This layer 2 encoder is referred to as a serial interface (SI), but does not have to be a serial port. This interoperation between PPP and the SI is shown in Figure C-3. The functions shown in the dotted rectangle are those provided by the serial interface software.

Figure C-3. Serial Interface (SI) Abstraction



As shown in Figure C-3, the SI interface has the responsibility of providing for connection control, a timer

used by PPP for timeout, packet encoding and decoding, and a SI callback function for status messages and packet transmission. Note that the SI driver developer also defines the actual API used by the application software to establish and tear down PPP connection sessions. There is no specific requirements in specifying the session API for any particular PPP device, but the APIs defined for HDLC and PPPoE can be used as a guide.

C.1.2 Function Overview

The SI interface module is charged with communicating with both the hardware and the application program, but the PPP packets themselves are processed via the PPP support functions in the stack. The PPP support software provides the following functions for use by the SI module:

pppNew()	Create a new PPP connection instance
pppFree()	Destroy an existing PPP connection instance
pppTimer()	Inform PPP that a 1 second timer tick has expired
pppInput()	Pass in a received PPP packet for processing

The formal declaration of these functions appear later in this section (see Section C.1.6).

Note: These functions can only be called in kernel mode. See Appendix for programming in kernel mode.

C.1.3 Supported Protocols

In keeping with trying to maintain a small footprint, the PPP software supports a subset of the general PPP protocols. The following are supported:

- Link Control Protocol (LCP)
- Internet Protocol Control Protocol (IPCP)
- Password Authentication Protocol (PAP)
- Challenge Handshake Authentication Protocol (CHAP) using MD5
- Internet Protocol (IP)

C.1.4 SI Module Callback Function

The PPP support API is used for connection instance creation and destruction, and to pass received packets to the stack. To get information about PPP back from the stack, and to allow the stack to request the transmission of PPP packets, the SI module supplies a callback function. A pointer to this callback is passed to PPP as a parameter to *pppNew()*.

Note: This function is called in kernel mode. See Appendix for programming in kernel mode.

C.1.4.1 Function Declaration

The SI callback function is provided in the SI code module using the following definition:

SIControl — ***Notify the Serial Interface of a Change in Status, or when SI Needs to Transmit a Packet***

Syntax

void SIControl(HANDLE hSI, uint Message, UINT32 Data, PBM_Handle hPkt);

Parameters

hSI	Handle to SI private data
Message	Message code describing the PPP event
Data	Additional data concerning the message
hPkt	Handle to a PBM packet when Message is SI_MSG_SENDPACKET

Return Value None.

Description This function is called when a PPP needs to notify the serial interface (SI) of a change in status, or when it needs SI to transmit a packet.

The *hSI* parameter is a handle (pointer to a void) that is originally passed to PPP via *pppNew()*. This value allows the SI module to know which of its own connection instances is in use. The PPP instance handle in use is not supplied, but rather should be obtained by reference from the supplied SI handle. If the programmer of the SI module does not wish to track handles, then this parameter may be NULL (always as originally supplied to *pppNew()*). This is NOT the handle to the PPP instance that is passed to other functions in the PPP API.

The purpose of the callback is determined by the value of the *Message* parameter. The following message values are defined for this parameter:

SI_MSG_CALLSTATUS	PPP connection status has changed
SI_MSG_SENDPACKET	PPP is requesting a packet to be encoded and transmitted
SI_MSG_PEERCMAP	LCP has received the peer's 32 bit asynchronous character map

C.1.4.2 SI_MSG_CALLSTATUS Message

When this message value is set, the callback function was called by PPP to update the status of the connection instance. When the callback is called with this message, the value of *Data* contains additional information about the call. *Data* can be set to any of the following values:

SI_CSTATUS_WAITING	Connection instance is idle
SI_CSTATUS_NEGOTIATE	Instance in LCP negotiation stage
SI_CSTATUS_AUTHORIZE	Instance in authorization stage
SI_CSTATUS_CONFIGURE	Instance in IP configuration stage
SI_CSTATUS_CONNECTED	Instance is fully connected and operational
SI_CSTATUS_DISCONNECT	Connection dropped
SI_CSTATUS_DISCONNECT_LCP	Connection dropped in LCP stage
SI_CSTATUS_DISCONNECT_AUTH	Connection dropped in authorization stage
SI_CSTATUS_DISCONNECT_IPCP	Connection dropped in IP configuration stage

In the case that *Data* is set to any of disconnect messages, *pppFree()* should be called to destroy the connection instance. For all other status values, no action is required.

Note: It is always safe to assume that when the value of Data >= SI_CSTATUS_DISCONNECT, the message is some type of disconnect.

C.1.4.3 SI_MSG_ SENDPACKET Message

When this message value is set, the callback function was called by PPP to transmit a packet. The *Data* parameter is set to the 16 bit PPP protocol of the packet, and the *hPkt* parameter contains a handle to a packet (PKT) object that contains the packet payload. It is the job of the SI callback function to encode the packet and transmit it on the physical hardware.

C.1.4.4 SI_MSG_ PEERCMAP Message

Serial interfaces to PPP require a translation map for the first 32 character values. This map informs the packet encoded which characters must be escaped and which do not. The default value of the peer CMAP should be 0xffffffff, and updated only when this message is received. Whether or not PPP will attempt to exchange CMAP information with its peer, is determined by passing flags to *pppNew()* when the connection instance is created.

C.1.4.5 Example Callback Function Implementation

The following is an example of a SI module callback function from the HDLC module code in the example applications. The code illustrates the basic processing that must be done for the various SI callback messages. The function calls made in this example are described in Appendix .

```
//---------------------------------------------------------------------
// SI Control Function
//---------------------------------------------------------------------
void hdlcSI( HANDLE hSI, uint Msg, UINT32 Aux, PBM_Handle hPkt )
{
    HDLC_INSTANCE     *pi = (HDLC_INSTANCE *)hSI;
    HANDLE            hTmp;
    uint              Offset,Size;
    UINT8             *pBuf;

    switch(Msg)
    {
    case SI_MSG_CALLSTATUS:
        // Update Connection Status
        pi->Status = (uint)Aux;
        if( Aux >= SI_CSTATUS_DISCONNECT )
        {
            // Close PPP
            if( pi->hPPP )
            {
                hTmp = pi->hPPP;
                pi->hPPP = 0;
                pppFree( hTmp );
            }
        }
        break;

    case SI_MSG_PEERCMAP:
        // Update Out CMAP for Transmit
        pi->cmap_out = Aux;
        llSerialHDLCPeerMap( pi->DevSerial, Aux );
        break;

    case SI_MSG_SENDPACKET:
        if( !hPkt )
        {
            DbgPrintf( DBG_ERROR,"hdlcSI: No packet" );
            break;
        }

        Offset = PBM_getDataOffset( hPkt );
        Size = PBM_getValidLen( hPkt );

        // Make sure packet is valid, with room for protocol, room for checksum
```

```
            if((Offset<4) || ((Offset+Size+2)>PBM_getBufferLen(hPkt)))
            {
                DbgPrintf( DBG_ERROR,"hdlcSI: Bad packet" );
                PBM_free( hPkt );
                break;

            }

            // Add in 2 byte Protocol and 2 byte header. Also add in size for
            // 2 byte checksum. Note that the outgoing checksum is corrected
            // (calculated) by the serial driver.
            Offset -= 4;
            Size += 6;
            PBM_setDataOffset(hPkt, Offset);
            PBM_setValidLen(hPkt, Size);
            pBuf = PBM_getDataBuffer(hPkt)+Offset;
            *pBuf++ = 0xFF;
            *pBuf++ = 0x03;
            *pBuf++ = (UINT8)(Aux/256);
            *pBuf = (UINT8)(Aux%256);

            // Send the buffer to the serial driver
            llSerialSendPkt(pi->DevSerial, hPkt);
            break;
        }
    }
```

C.1.5 Tips for Implementing a PPP Serial Interface (SI) Module Instance

C.1.5.1 Multiple Instances

PPP supports multiple instances, but the SI module implementation tracks multiple instances of itself. This is done in two ways. One method is for the SI module to have a locally global head pointer to its first instance, and an array or linked list for additional instances. Or, the instance can be bound to the next layer down. In the case of the HDLC module, one PPP instance is bound to one serial port driver instance. So the HDLC module does not need to track instances independently.

When a new PPP connection is established, a new SI module instance should be allocated and a handle to the new SI instance is passed to the *pppNew()* function. The handle that *pppNew()* returns must be associated with the handle to the SI instance. The PPP handle must be passed to all other PPP API functions, and PPP will pass back the SI instance handle to the SI callback function.

When new data arrives from the hardware, it is the responsibility of the SI module to associate that data with a specific SI instance. The SI instance can then be accessed to retrieve the handle to the PPP instance to use with any PPP function calls. In the case of HDLC, the SI instance is known because it is associated with a particular serial device instance.

C.1.5.2 Using the Timer Object

PPP requires that its *pppTimer()* function be called once every second. This can be PRD driven if necessary, but the timer callback cannot be called from a PRD because it must be called from within kernel mode (an *llEnter()*/*llExit()*) pairing.

C.1.5.3 Registering Packet Padding Requirements

Although a serial interface will probably not have any special requirements for packets from the stack, it must at least be able to construct valid packets to send to the *pppInput()* function. To use the packet allocation function provided by the IF API (see Appendix), the SI module should declare its padding requirements via the *IFSetPad()* function. For a serial interface that does not use the packet buffer to physically send the packet, the size of the PPP header would be 4 bytes (2 byte HDLC header and 2 byte protocol field), and the padding would be 2 bytes (checksum).

C.1.6 PPP API Functions

The following is the full description of the PPP functions described in this section.

pppNew — *Create a New PPP Connection Instance*

Syntax

HANDLE pppNew(HANDLE hSI, uint pppFlags, uint mru, IPN IPServer, IPN IPMask, IPN IPClient, char *Username, char *Password, UINT32 cmap, void (*pfnSICtrl)(HANDLE, uint, UINT32, HANDLE));

Parameters

hSI	Handle to SI module to be passed back to callback function
pppFlags	Connection option flags
mru	Maximum receive unit (maximum size of Payload)
IPServer	IP address of server in server mode (NULL in client mode)
IPMask	IP subnet mask of client in server mode (NULL in client mode)
IPClient	IP address of client in server mode (NULL in client mode)
Username	Pointer to username in client mode (NULL in server mode)
Password	Pointer to password in client mode (NULL in server mode)
cmap	32-bit local CMAP to pass to peer
pfnSICtrl	Pointer to SI module callback function

Return Value

Handle to new PPP connection instance, or NULL on error.

Description

This function is called to create a new PPP connection instance. The type of connection created is determined by the calling parameters.

- *hSI* - This is a private handle created by the caller that points back to the caller's instance data. It is passed back to the caller via the callback function pointed to by *pfnSICtrl*, and can be used to link back to caller's instance data when the callback is executed.
- *pppFlags* - The flags determine what type of connection instance to create, and what type of options to support in the connection instance. In the *pppFlags* parameter, one and only one of the following flags must be set:

PPPFLG_SERVER	Create PPP server connection instance
PPPFLG_CLIENT	Create PPP client connection instance

When operating in SERVER mode, any of the following flags can also be set:

PPPFLG_OPT_AUTH_PAP	Require PAP authentication
PPPFLG_OPT_AUTH_CHAP	Require CHAP authentication
PPPFLG_OPT_USE_MSE	Use MS extensions as server
PPPFLG_OPT_LOCALDNS	Claim Local IP as DNS server
PPPFLG_SIOPT_SENDCMAP	Send an async character map
PPPFLG_SIOPT_RECVCMAP	Accept an async character map
PPPFLG_CH1	Allow server channel/group 1 account users
PPPFLG_CH2	Allow server channel/group 2 account users
PPPFLG_CH3	Allow server channel/group 3 account users
PPPFLG_CH4	Allow server channel/group 4 account users

PPPFLG_OPT_ALLOW_IP	Allow client to declare its own IP address
PPPFLG_OPT_ALLOW_HC	Allow peer to negotiate PFC/ACFP

When operating in CLIENT mode, any of the following flags can also be set:

PPPFLG_OPT_USE_MSE	Use MS extensions as client
PPPFLG_OPT_CLIENT_P2P	Treat the connection as a pure peer to peer (i.e., do not create a default route using the peer as a gateway)
PPPFLG_SIOPT_SENDCMAP	Send an async character map
PPPFLG_SIOPT_RECVCMAP	Accept an async character map
PPPFLG_OPT_ALLOW_HC	Allow peer to negotiate PFC/ACFP

- *mru* - The MRU is maximum receive unit, or the maximum size of the payload portion of a PPP packet. For a standard serial link, the MRU is typically 1500, but can be smaller.
- *IPServer* - When creating the PPP instance in SERVER mode, this is the IP address in network format of the NDK reported to the peer. When operating in CLIENT mode, this value is NULL.
- *IPMask* - When creating the PPP instance in SERVER mode, this is the IP subnet mask of the peer's IP network reported to the peer. When operating in CLIENT mode, this value is NULL.
- *IPClient* - When creating the PPP instance in SERVER mode, this is the IP address in network format of the peer reported to the peer. When operating in CLIENT mode, this value is NULL.
- *Username* - When creating the PPP instance in CLIENT mode, this is a pointer to a NULL terminated string containing the username to use in PAP or CHAP authentication. The maximum string length is defined by PPPNAMELEN. When operating in SERVER mode, this value is NULL.
- *Password* - When creating the PPP instance in CLIENT mode, this is a pointer to a NULL terminated string containing the password to use in PAP or CHAP authentication. The maximum string length is defined by PPPNAMELEN. When operating in SERVER mode, this value is NULL.
- *cmap* - When the PPPFLG_SIOPT_SENDCMAP flag is set in the pppFlags parameter, this is the CMAP value that is sent to the peer; otherwise it is NULL.
- *pfnSICtrl* - This is a required pointer to the caller's callback function to handle status updates from the stack, and requests to transmit PPP packets. See Section C.1.4 for more detail.

When run in SERVER mode, the name of the PPP server defaults to DSPIP in CHAP authentication; however, this can be changed by using the CFGITEM_SYSINFO_REALMPPP configuration tag. For example:

```
// Name our authentication group for PPP (Max size = 31)
// This is the authentication "realm" name returned by the PPP
// server when authentication is required.
// (Note the length "16" includes the NULL terminator)

CfgAddEntry( hCfg, CFGTAG_SYSINFO, CFGITEM_SYSINFO_REALMPPP,
             0, 16, (UINT8 *)"PPP_SAMPLE_NAME", 0 );
```

When successful, this function returns a handle to a new PPP instance. This handle is used by the caller when calling other functions in the PPP API.

pppFree — *Destroy PPP Connection Instance*

Syntax

void pppFree(HANDLE hPPP);

Parameters

hPPP	Handle to PPP instance created with pppNew()

Return Value None.

Description This function is called to close and destroy a PPP connection instance created with *pppNew()*. This function must be called to free the PPP handle, even if the PPP connection itself is already disconnected.

pppInput *Send a PPP Packet to PPP for Processing*

Syntax void pppInput(HANDLE hPPP, PBM_Pkt *pPkt);

Parameters

hPPP	Handle to PPP instance created with *pppNew()*
pPkt	Pointer to a PBM packet

Return Value None.

Description This function is called when a PPP packet is received on a active serial interface. The packet is data decoded into the PPP protocol and payload fields, and given to PPP as a packet object. The handle *hPPP* is the PPP connection instance returned from *pppNew()* for this connection, and *pPkt* is a packet object created by the packet buffer manager (PBM).

pppTimer *Notify PPP of One Second Tick*

Syntax void pppTimer(HANDLE hPPP);

Parameters

hPPP	Handle to PPP instance created with *pppNew()*

Return Value None.

Description This function is called on an active PPP instance to notify PPP that one second has elapsed. Because the PPP API is entirely stateless, it relies on the serial interface for time tick notification.

C.2 Serial HDLC Client and Server Support

Note: The HDLC API is user-callable. Unlike the low level PPP support API, you should *not* use the *llEnter()/llExit()* functions when calling the functions described in this section.

C.2.1 Synopsis

This implementation of HDLC for the NDK library is included in the example applications. It interfaces to the serial port driver described in the HAL.

C.2.2 Function Overview

Called by Application:

hdlcNew()	Create a Serial HDLC Client Session
hdlcFree()	Destroy a Serial HDLC Client Session

hdlcGetStatus()	Get the Call Status of a Serial HDLC Client Session
hdlcsNew()	Create a Serial HDLC Server Session
hdlcsFree()	Destroy a Serial HDLC Server Session
hdlcsGetStatus()	Get the Call Status of a Server HDLC Client Session

Called by Serial Port Driver:

hdlcInput()	Send HDLC input buffer for processing

C.2.3 HDLC API Functions

hdlcNew — *Create a Serial HDLC Client Session*

Syntax

HANDLE hdlcNew(uint Dev, uint pppFlags, UINT32 cmap, char *Username, char *Password);

Parameters

Dev	Physical index of serial port to use
pppFlags	Connection option flags
cmap	Async control character map
Username	Pointer to client account username
Password	Pointer to client account password

Return Value

If it succeeds, the function returns a handle to a HDLC client instance. Otherwise, it returns NULL.

Description

This function is called to create a new serial HDLC client instance on the physical serial interface specified by the index *Dev*.

- *pppFlags* - The flags determine what type of connection instance to create, and what type of options to support in the connection instance. In the *pppFlags* parameter, the following flag must be set:

PPPFLG_CLIENT	Create PPP client connection instance

  In addition, any of the following flags can also be set:

PPPFLG_OPT_USE_MSE	Use MS extensions as client
PPPFLG_OPT_CLIENT_P2P	Treat the connection as a pure peer to peer (i.e., don't create a default route using the peer as a gateway).
PPPFLG_SIOPT_SENDCMAP	Send an async character map (strongly recommended)
PPPFLG_SIOPT_RECVCMAP	Accept an async character map (strongly recommended)
PPPFLG_OPT_ALLOW_HC	Allow peer to negotiate PFC/ACFP

- *cmap* - This is the desired value of the async character control map that is sent to the peer to allow frame compression by skipping the escape coding of characters when it is not required. The mask contains a set bit for each character (0 to 31) that *must* be escaped when sent by the peer. If the PPPFLG_SIOPT_SENDCMAP option is not set, it is assumed that all 32 characters must be sent via the escape sequence.
- *Username* - This is a pointer to a NULL terminated string containing the username to use in PAP or CHAP authentication. The maximum string length is defined by PPPNAMELEN.
- *Password* - This is a pointer to a NULL terminated string containing the password to use in PAP or CHAP authentication. The maximum string length is defined by PPPNAMELEN.

When successful, this function returns a handle to a new serial HDLC instance. The current status of the connection can be queried at any time by calling *hdlcGetStatus()*.

hdlcFree — *Destroy a Serial HDLC Client Session*

Syntax

void hdlcFree(HANDLE hHDLC);

Parameters

hHDLC	Handle to HDLC Client Session

Return Value None.

Description This function is called to close and destroy a serial HDLC client session that was created with *hdlcNew()*. This function is always called once for every HDLC instance handle. If the connection is no longer active, it frees the instance memory. If the connection is still active, it disconnects the call first.

hdlcGetStatus — *Get the Status of a Serial HDLC Client Session*

Syntax uint hdlcGetStatus(HANDLE hHDLC);

Parameters

hHDLC	Handle to HDLC Client Session

Return Value This function returns a uint that will be set to one of the following values:

SI_CSTATUS_WAITING	Connection is idle (HDLC session opening)
SI_CSTATUS_NEGOTIATE	Connection in LCP negotiation stage
SI_CSTATUS_AUTHORIZE	Connection in authorization stage
SI_CSTATUS_CONFIGURE	Connection in IP configuration stage
SI_CSTATUS_CONNECTED	Connection is fully connected and operational
SI_CSTATUS_DISCONNECT	Connection dropped
SI_CSTATUS_DISCONNECT_LCP	Connection dropped in LCP stage
SI_CSTATUS_DISCONNECT_AUTH	Connection dropped in authorization stage
SI_CSTATUS_DISCONNECT_IPCP	Connection dropped in IP configuration stage

Description This function is called to get the connection status of a serial HDLC client session using the HDLC instance handle returned from *hdlcNew()*. This function can be called anytime after the handle is created with *hdlcNew()*, and before it is destroyed with *hdlcFree()*.

hdlcsNew — *Create a Serial HDLC Server Session*

Syntax HANDLE hdlcsNew(uint Dev, uint pppFlags, UINT32 cmap, IPN IPServer, IPN IPMask, IPN IPClient);

Parameters

Dev	Physical index of serial port to use
pppFlags	Connection option flags
cmap	Async control character map
IPServer	IP address of server in network format
IPMask	IP subnet mask in network format of the peer's network
IPClient	IP address in network format of the client

Return Value If it succeeds, the function returns a handle to a serial HDLC server instance. Otherwise, it returns NULL.

Description This function is called to create a new serial HDLC server instance on the physical serial interface specified by the index *Dev*.

- *pppFlags* - The flags determine what type of connection instance to create, and what

type of options to support in the connection instance. In the *pppFlags* parameter, the following flag must be set:

PPPFLG_SERVER	Create PPP server connection instance

In addition, any of the following flags can also be set:

PPPFLG_OPT_AUTH_PAP	Require PAP authentication
PPPFLG_OPT_AUTH_CHAP	Require CHAP authentication (PAP is fallback when specified)
PPPFLG_OPT_USE_MSE	Use MS extensions as server
PPPFLG_SIOPT_SENDCMAP	Send an async character map (*strongly* recommended)
PPPFLG_SIOPT_RECVCMAP	Accept an async character map (*strongly* recommended)
PPPFLG_CH1	Allow server channel/group 1 account users
PPPFLG_CH2	Allow server channel/group 2 account users
PPPFLG_CH3	Allow server channel/group 3 account users
PPPFLG_CH4	Allow server channel/group 4 account users
PPPFLG_OPT_ALLOW_IP	Allow client to declare its own IP address
PPPFLG_OPT_ALLOW_HC	Allow peer to negotiate PFC/ACFP

- *cmap* - This is the desired value of the async character control map that is sent to the peer to allow frame compression by skipping the escape coding of characters when it is not required. The mask contains a set bit for each character (0 to 31) that *must* be escaped when sent by the peer. If the PPPFLG_SIOPT_SENDCMAP option is not set, it is assumed that all 32 characters must be sent via the escape sequence.
- *IPServer* - This is the IP address in network format of the NDK reported to the peer.
- *IPMask* - This is the IP subnet mask of the peer's IP network reported to the peer.
- *IPClient* - This is the IP base address in network format of the IP address to be assigned to the client.

When successful, this function returns a handle to a new serial HDLC server instance. The current status of the connection can be queried at any time by calling *hdlcsGetStatus()*.

The name of the PPP server defaults to DSPIP in CHAP authentication; however, this can be changed by using the CFGITEM_SYSINFO_REALMPPP configuration tag. For example:

```
// Name our authentication group for PPP (Max size = 31)
// This is the authentication "realm" name returned by the PPP
// server when authentication is required.
// (Note the length "16" includes the NULL terminator)

CfgAddEntry( hCfg, CFGTAG_SYSINFO, CFGITEM_SYSINFO_REALMPPP,
             0, 16, (UINT8 *)"PPP_SAMPLE_NAME", 0 );
```

hdlcsFree — *Destroy a Serial HDLC Server Session*

Syntax

void hdlcsFree(HANDLE hHDLC);

Parameters

	hHDLC	Handle to HDLC Server Session

Return Value None.

Description This function is called to close and destroy a serial HDLC server session that was created with *hdlcsNew()*. This function is always called once for every HDLC instance handle. If the connection is no longer active, it frees the instance memory. If the connection is still active, it disconnects the call first.

hdlcsGetStatus *Get the Status of a Serial HDLC Server Session*

Syntax uint hdlcsGetStatus(HANDLE hHDLC);

Parameters

	hHDLC	HDLC Server Session

Return Value This function returns a uint that will be set to one of the following values:

SI_CSTATUS_WAITING	Connection is idle (PPPoE session opening)
SI_CSTATUS_NEGOTIATE	Connection in LCP negotiation stage
SI_CSTATUS_AUTHORIZE	Connection in authorization stage
SI_CSTATUS_CONFIGURE	Connection in IP configuration stage
SI_CSTATUS_CONNECTED	Connection is fully connected and operational
SI_CSTATUS_DISCONNECT	Connection dropped
SI_CSTATUS_DISCONNECT_LCP	Connection dropped in LCP stage
SI_CSTATUS_DISCONNECT_AUTH	Connection dropped in authorization stage
SI_CSTATUS_DISCONNECT_IPCP	Connection dropped in IP configuration stage

Description This function is called to get the connection status of a serial HDLC server session using the HDLC instance handle returned from *hdlcsNew()*. This function can be called anytime after the handle is created with *hdlcsNew()*, and before it is destroyed with *hdlcsFree()*.

C.3 PPPoE Client and Server Support

Note: The PPPoE API is user callable. Unlike the low level PPP support API, you should *not* use the *llEnter()/llExit()* functions when calling the functions described in this section.

C.3.1 Synopsis

The PPPoE (PPP over Ethernet) specification allows for PPP packets to be transmitted in a peer to peer method over an Ethernet tunnel. The standard has gained in popularity because it allows for the use of multiple user accounts on a single Ethernet network.

The implementation of PPPoE supplied in the NDK library is built into the stack library code, and linked to the Ether object that handles packets from all Ethernet devices in the HAL layer. Thus, is it not necessary to access or alter the HAL to use PPPoE.

The software can be used as a PPP server or PPP client, but not both simultaneously. In both cases, PPPoE uses the the PPP programming interfaces described earlier in this section. Thus, for server mode, the PPP server will use the same user account information as a serial based server.

C.3.2 Function Overview

The PPPoE function API is short:

pppoeNew()	Create a PPPoE Client Session
pppoeFree()	Destroy a PPPoE Client Session
pppoeGetStatus()	Get the Call Status of a PPPoE Client Session
pppoesNew()	Create a PPPoE Server Session
pppoesFree()	Terminate a PPPoE Server Session

C.3.3 PPPoE API Functions

pppoeNew — ***Create a PPPoE Client Session***

Syntax

HANDLE pppoeNew(HANDLE hEther, uint pppFlags, INT8 *Username, INT8 *Password);

Parameters

hEther	Handle to Ether device on which to look for a PPPoE server
pppFlags	Connection option flags
Username	Pointer to client account username
Password	Pointer to client account password

Return Value

If it succeeds, the function returns a handle to a PPPoE client instance. Otherwise, it returns NULL.

Description

This function is called to create a new PPPoE client instance on the Ether type interface specified by the handle *hEther*.

- *pppFlags* - The flags determine what type of connection instance to create, and what type of options to support in the connection instance. In the *pppFlags* parameter, the following flag must be set:

PPPFLG_CLIENT	Create PPP client connection instance

  In addition, any of the following flags can also be set:

PPPFLG_OPT_USE_MSE	Use MS extensions as client
PPPFLG_OPT_CLIENT_P2P	Treat the connection as a pure peer to peer (i.e., do not create a default route using the peer as a gateway)
PPPFLG_OPT_ALLOW_HC	Allow peer to negotiate PFC/ACFP

- *Username* - This is a pointer to a NULL terminated string containing the username to use in PAP or CHAP authentication. The maximum string length is defined by PPPNAMELEN.
- *Password* - This is a pointer to a NULL terminated string containing the password to use in PAP or CHAP authentication. The maximum string length is defined by PPPNAMELEN.

When successful, this function returns a handle to a new PPPoE instance The current status of the PPPoE connection can be queried at any time by calling *pppoeGetStatus()*.

pppoeFree — *Destroy a PPPoE Client Session*

Syntax

void pppoeFree(HANDLE hPPPOE);

Parameters

hPPPOE	Handle to PPPoE Client Session

Return Value

None.

Description

This function is called to close and destroy a PPPoE client session that was created with *pppoeNew()*. This function is always called once for every PPPoE instance handle. If the connection is no longer active, it frees the instance memory. If the connection is still active, it first disconnects the call.

pppoeGetStatus — *Get the Status of a PPPoE Client Session*

Syntax

uint pppoeGetStatus(HANDLE hPPPOE);

Parameters

hPPPOE	Handle to PPPoE Client Session

Return Value

This function returns a uint that will be set to one of the following values:

SI_CSTATUS_WAITING	Connection is idle (PPPoE session opening)
SI_CSTATUS_NEGOTIATE	Connection in LCP negotiation stage
SI_CSTATUS_AUTHORIZE	Connection in authorization stage
SI_CSTATUS_CONFIGURE	Connection in IP configuration stage
SI_CSTATUS_CONNECTED	Connection is fully connected and operational
SI_CSTATUS_DISCONNECT	Connection dropped
SI_CSTATUS_DISCONNECT_LCP	Connection dropped in LCP stage
SI_CSTATUS_DISCONNECT_AUTH	Connection dropped in authorization stage
SI_CSTATUS_DISCONNECT_IPCP	Connection dropped in IP configuration stage

Description

This function is called to get the connection status of a PPPoE client session using the PPPoE instance handle returned from *pppoeNew()*. This function can be called anytime after the handle is created with *pppoeNew()*, and before it is destroyed with *pppoeFree()*.

pppoesNew — *Create a PPPoE Server Session*

Syntax

HANDLE pppoesNew(HANDLE hEther, uint pppFlags, uint SessionMax, IPN IPServer, IPN IPMask, IPN IPClientBase, INT8 *ServerName, INT8 *ServiceName);

Parameters

hEther	Handle to Ether device on which to invoke the PPPoE server
pppFlags	Connection option flags
SessionMax	Maximum number of client connections allowed
IPServer	IP address of server in network format
IPMask	IP subnet mask in network format of the client address pool
IPClientBase	IP base address in network format of the client address pool
ServerName	Server name reported via PPPoE protocol

ServiceName	Service name reported via PPPoE protocol

Return Value

If it succeeds, the function returns a handle to a PPPoE server instance. Otherwise, it returns NULL.

Description

This function is called to create a new PPPoE server instance on the Ether type interface specified by the handle *hEther*.

- *SessionMax* - This value is the maximum number of simultaneous peer connections to be allowed at any given time. Thus, it is also the minimum size of the client IP address pool.
- *pppFlags* - The flags determine what type of connection instance to create, and what type of options to support in the connection instance. In the *pppFlags* parameter, the following flag must be set:

PPPFLG_SERVER	Create PPP server connection instance

  In addition, any of the following flags can also be set:

PPPFLG_OPT_AUTH_PAP	Require PAP authentication
PPPFLG_OPT_AUTH_CHAP	Require CHAP authentication
PPPFLG_OPT_USE_MSE	Use MS extensions as server
PPPFLG_OPT_LOCALDNS	Claim Local IP as DNS server
PPPFLG_CH1	Allow server channel/group 1 account users
PPPFLG_CH2	Allow server channel/group 2 account users
PPPFLG_CH3	Allow server channel/group 3 account users
PPPFLG_CH4	Allow server channel/group 4 account users
PPPFLG_OPT_ALLOW_IP	Allow client to declare its own IP address
PPPFLG_OPT_ALLOW_HC	Allow peer to negotiate PFC/ACFP

- *IPServer* - This is the IP address in network format of the NDK reported to the peer.
- *IPMask* - This is the IP subnet mask of the peer's IP network reported to the peer.
- *IPClientBase* - This is the IP base address in network format of the IP address pool to be assigned to and reported to peer connections. The size of the address pool is determined by the value of SessionMax.
- *ServerName* - This is a required pointer to a NULL terminated string containing the server name that is reported to PPPoE clients. The maximum length of this name including the NULL terminator is defined by PPPOE_NAMESIZE. If a longer name is supplied, this function will fail.
- *ServiceName* - This is a required pointer to a NULL terminated string containing the service name that is reported to PPPoE clients. The maximum length of this name, including the NULL terminator, is defined by PPPOE_NAMESIZE. If a longer name is supplied, this function will fail.

  The name of the PPP server defaults to DSPIP in CHAP authentication. This is independent of the PPPoE server name. However, the name can be changed by using the CFGITEM_SYSINFO_REALMPPP configuration tag. For example:

```
// Name our authentication group for PPP (Max size = 31)
// This is the authentication "realm" name returned by the PPP
// server when authentication is required.
// (Note the length "16" includes the NULL terminator)

CfgAddEntry( hCfg, CFGTAG_SYSINFO, CFGITEM_SYSINFO_REALMPPP,
             0, 16, (UINT8 *)"PPP_SAMPLE_NAME", 0 );
```

When successful, this function returns a handle to a new PPPoE server instance. The status of individual connections is not available to the caller, but tracked automatically by PPPoE. When sessions are added or destroyed, the IP address callback supplied to

NC_NetStart() is called and connections can be tracked by the applications programmer via this function callback.

pppoesFree — *Destroy a PPPoE Server Session*

Syntax

void pppoesFree(HANDLE hPPPOES);

Parameters

hPPPOES	Handle to PPPoE Server Session

Return Value

None.

Description

This function is called to close and destroy a PPPoE server session that was created with *pppoesNew()*. This function is always called once to shut down the PPPoE server. Any external client currently connected to the server is disconnected.

C.4 Creating PPP Server User Accounts

C.4.1 Synopsis

To use the PPP or PPPoE protocol in server mode, it advisable to protect access to the system through the use of a PPP authentication protocol. The PPP supplied in the stack library allows for the use of either PAP or CHAP in user authentication. The database of authorized users (name and password) is stored in the configuration system.

C.4.2 Adding and Reviewing User Accounts

The definition of the user account entry in the configuration system is defined in Section 4.4.7. Note in that section that the server channel flags PPPFLG_CH1 through PPPFLG_CH4 are duplicated in both the server flags and the client account flags. This allows the system programmer to allow different classes of services for different channels.

The methodology of adding, querying, and removing user accounts is the same for any other tag in the configuration system. Some simple examples follow. More example code can be found in the sample console program.

C.4.2.1 Adding a PPP User Account

The following code adds a PPP user account for the user supplied in *name* with a password supplied in *password*. Note that it also uses the *AcctFind()* function to verify that the account does not already exist.

```
void AcctAdd(char *name, char *password)
{
    CI_ACCT CA;
    HANDLE hAcct;
    int rc;

    // Check string lengths for name and password
    if(strlen(name) >= CFG_ACCTSTR_MAX ||
        strlen(password) >= CFG_ACCTSTR_MAX)
    {
        printf("Name or password too long, %d character max\n\n",
                  CFG_ACCTSTR_MAX-1);
        return;
    }

    // See if the account already exists
    hAcct = AcctFind(tok2);
    if(hAcct)
    {
        printf("Account exits - remove old account first\n\n");
```

```
        // We must de-reference the account we found
        CfgEntryDeRef(hAcct);
        return;
    }

    // Fill in the CA record
    strcpy(CA.Username, name);
    strcpy(CA.Password, password);

    // Give user access to all channels
    CA.Flags =
CFG_ACCTFLG_CH1|CFG_ACCTFLG_CH2|CFG_ACCTFLG_CH3|CFG_ACCTFLG_CH4;

    // Add it to the configuration
    rc = CfgAddEntry(0, CFGTAG_ACCT, CFGITEM_ACCT_PPP,
                     CFG_ADDMODE_NOSAVE, sizeof(CI_ACCT),
                     (UINT8 *)&CA, 0);

    if(rc < 0)
        printf("Error adding account\n");
    else
        printf("Account added\n");

    return;
}
```

C.4.2.2 Searching for a PPP User Account

The following code implements the *AcctFind()* function called in the previous example. Note that the same method could be used to print out a list of all accounts.

```
HANDLE AcctFind(char *name)
{
    HANDLE hAcct;
    CI_ACCT CA;
    int rc;
    int size;

    // Get the first user account
    rc = CfgGetEntry(0, CFGTAG_ACCT, CFGITEM_ACCT_PPP, 1, &hAcct);

    // If there are no accounts, then we did not find it
    if(rc <= 0)
        return(0);

    // Search until we run out of accounts or have a match
    while(1)
    {
        // Get the data for this entry into CA
        size = sizeof(CA);
        rc = CfgEntryGetData(hAcct, &size, (UINT8 *)&CA);
        if(rc <= 0)
        {
            // This is an unexpected error - deref the handle and abort
            CfgEntryDeRef(hAcct);
            return(0);
        }

        // See if the username matches the search name. If so, return
        // the referenced handle
        if(!strcmp(name, CA.Username))
            return(hAcct);

        // Since we did not match, get the next entry. If there is no
        // next entry, we are done searching.
```

```
            rc = CfgGetNextEntry(0, hAcct, &hAcct);
            if(rc <= 0)
                return(0);
        }
    }
```

C.4.2.3 Removing a PPP User Account

Removing a specific user account is done by finding the account and removing the entry handle.

The following uses the *AcctFind()* function to find the target account.

```
void AcctDelete(char *name)
{
    HANDLE hAcct;

    // Find the account to delete
    hAcct = AcctFind(name);

    // If we found the account, remove it
    if(hAcct)
    {
        CfgRemoveEntry(0, hAcct);
        printf("Account removed\n");
    }
}
```

Hardware Adaptation Layer (HAL)

As discussed in the introduction, hardware devices are supported through a Hardware Adaptation Layer. This section describes the HAL API.

This section is required only for system programming that needs low level access to the hardware for configuration and monitoring. **This API does not apply to sockets application programming.**

Topic **Page**

D.1 Overview

The function of the HAL is to provide resources to the stack library functions and allow them to operate independently of the current run-time environment. The HAL contains the functionality required by the stack that depends directly on the hardware in a particular environment.

D.1.1 HAL Function Types

The HAL is interspersed with two different types of functions; those that are called at kernel level (inside an *llEnter()/llExit()* pairing), and those that are not. (For more information on the *llEnter()* and *llExit()* functions, see Section A.1.)

To distinguish kernel level functions from application support functions, both have been given a different naming conventions. Kernel level functions are named with an ll prefix, without a leading underscore, for example: *llPacketSend()*, while application functions have an underscore, for example: *_llPacketInit()*.

D.1.2 External Calls from HAL Functions

Because HAL functions are called from the stack kernel, they are executing within an *llEnter()/llExit()* pair. These HAL functions can call the stack API directly, but should not call normal application functions.

If a HAL function must call an external application function, or if it is going to call a potentially blocking function, then it should first call *llExit()*. Then, when it has completed, it should call *llEnter()* before returning to the stack. **It is important not to block while in an *llEnter()/llExit()* pair.**

D.2 Low-Level LED Driver (llUserLed)

D.2.1 Synopsis

The User LED driver is not really a driver at all. It is a collection of functions to control (ON|OFF|TOGGLE) LED lights on a given hardware platform.

D.2.2 Function Overview

Application Functions:

_llUserLedInit()	Initialize the LED displays to their default state
_llUserLedShutdown()	Shut down the LED environment
LED_ON()	Turn on a LED
LED_OFF()	Turn off a LED
LED_TOGGLE()	Toggle the state of a LED

D.2.3 Low-Level LED API Functions

The following functions are required.

_llUserLedInit *Initialize the LED Displays to their Default State*

Syntax void _ llUserLedInit();

Return Value None.

Description This function initializes anything necessary to get the LED displays to their default state.

_llUserLedShutdown *Shutdown the LED Environment*

Syntax void _llUserLedShutdown();

Return Value None.

Description This function is called when shutting down the system to shut down and clean up the LED environment. Typically, this is an empty function.

LED_ON *Turn On an LED*

Syntax void LED_ON(UINT32 ledId);

Description This function turns on the specified LED in the calling argument.

LED_OFF *Turn Off an LED*

Syntax void LED_OFF(UINT32 ledId);

Description This function turns off the LED specified in the calling argument.

LED_TOGGLE *Toggle the State of an LED*

Syntax void LED_TOGGLE(UINT32 ledId);

Description This function toggles the on/off state of an LED specified in the calling argument.

D.3 Low-Level Timer Driver (llTimer)

D.3.1 Synopsis

The stack code requires a very basic simple time function. It consists of two parts: a function API, which can be called from the stack to get the current time, and a scheduler that sends timer event notifications every 100ms using the STKEVENT event object.

D.3.2 Function Overview

Application Functions:

_llTimerInit()	Initialize Timer Environment
_llTimerShutdown()	Shutdown Timer Environment

Kernel Layer Functions:

llTimerGetTime()	Get the Current Time
llTimerGetStartTime()	Get the Initial Startup Time

D.3.3 Low-Level Timer API Functions

The following functions are required.

_llTimerInit — ***Initialize Timer Environment***

Syntax void _llTimerInit(STKEVENT_Handle hEvent, UINT32 ctime);

Return Value None.

Description This function is called to initialize the timer environment, and to set the initial time. The value of *ctime* is the number of seconds elapsed from a known reference. An initial value of zero is also acceptable. The stack software is only tracks relative time. Take care when setting this value because the stack does not manage the timer value wrapping. This occurs every 136 years, or in 2116 if time is based off of Jan 1, 1980.

Every 100mS, the timer driver will indicate a timer event to the event object specified by *hEvent*. This STKEVENT object is discussed in Section A.4.

_llTimerShutdown — ***Shutdown Timer Environment***

Syntax void _llTimerShutdown();

Return Value None.

Description This function is called when shutting down the system, to shut down and clean up the timer environment.

llTimerGetTime *Get Current Time in Seconds and Milliseconds*

Syntax UINT32 llTimerGetTime(UINT32 *pMSFrac);

Description Returns the number of seconds that have elapsed since the timer driver was started. If the pointer *pMSFrac* is non-zero, the function writes the fractional seconds (in milliseconds) to this location (0 to 999).

> **Note:** Although the stack does not require real time, do not simply use a millisecond timer and divide by 1000, as the value will wrap every 50 days. Device drivers should attempt to provide a time value accurate down to millisecond granularity.

llTimerGetStartTime *Get the Initial Startup Time*

Syntax UINT32 llTimerGetStartTime();

Return Value Initial start time in seconds.

Description Returns the initial start time that was passed to *_llTimerOpen()*.

D.4 Low-Level Packet Driver (llPacket)

D.4.1 Synopsis

The stack code requires a very basic packet function library. Note that although the high level packet API is documented here, the HAL contains a generic packet driver that implements this API. It is more efficient to use the standard llPacket driver and provide a hardware specific mini-driver than to implement the llPacket API from scratch. The llPacket mini-driver is described in the support package documentation for your hardware platform (*TMS320C6000 Network Developer's Kit (NDK) Support Package for DSK6455 User's Guide* (SPRUES4) or *TMS320C6000 Network Developer's Kit (NDK) Support Package for EVMDM642 User's Guide* (SPRUES5)).

D.4.2 Function Overview

Application Functions:

_llPacketInit()	Initialize Driver Environment and Enumerate Devices
_llPacketShutdown ()	Shutdown Driver Environment
_llPacketServiceCheck()	Check for Packet Activity

Kernel Layer Functions:

llPacketOpen()	Open Driver and Bind Logical Ether Object to Device Id
llPacketClose()	Close Driver and Unbind Logical Ether Object from Device Id
llPacketSetRxFilter()	Set Packet Receive Filter
llPacketGetMacAddr()	Get MAC address
llPacketGetMCastMax()	Get the Maximum Number of Multicast Addresses
llPacketGetMCast()	Get Multicast Address List
llPacketSetMCast()	Set Multicast Address List
llPacketService()	Service a Queued Packet
llPacketSend()	Send a Packet

D.4.3 Low-Level Packet API Functions

The low-level support layer must provide the following functions:

_llPacketInit *Initialize Driver Environment and Enumerate Devices*

Syntax uint _llPacketInit(STKEVENT_Handle hEvent);

Return Value Returns the number of physical packet devices.

Description This function is called by NETCTRL to initialize the packet driver environment. This function also enumerates all the physical packet devices in the system, and returns a device count. The stack will then call the *llPacketOpen()* function once for each physical device indicated.

The *hEvent* calling parameter is a handle to a STKEVENT object that must be signaled whenever a packet is received. This STKEVENT object is discussed in Section A.4.

_llPacketShutdown *Shutdown Driver Environment*

Syntax void _llPacketShutdown();

Return Value None.

Description This function is called by NETCTRL to indicate a final shutdown of the packet driver environment. When called, there should be no currently open packet drivers, and *_llPacketInit()* will be called again before any call to *llPacketOpen()*.

_llPacketServiceCheck *Check for Ethernet Packet Activity*

Syntax void _llPacketServiceCheck(uint fTimerTick);

Return Value None.

Description This function is called by NETCTRL to check if packets are available from the Ethernet device. In a polling system, this function is called continuously. In an interrupt driven semaphore system, it is called when packet activity is indicated via the STKEVENT object, and also by the scheduler at 100ms timer intervals for dead man polling checks.

In both polling and interrupt environments, the *fTimerTick* flag will be set whenever a 100ms timer tick has occurred.

If any new packets are detected from within this function, the packet driver should signal the STKEVENT object in the passive mode (do not set the *fHwAsynch* flag in the *STKEVENT_signal()* function). This only applies to new packet events detected from within this function. The STKEVENT object is discussed in Section A.4.

llPacketOpen *Open Driver and Bind Logical Ether Object to Device ID*

Syntax uint llPacketOpen(uint dev, HANDLE hEther);

Return Value This function should return 1 on success, and 0 on failure.

Description Opens the low level packet driver specified by the one's based index *dev*. The maximum value of *dev* is the number of devices returned from the *_llPacketInit()* function. When opening the device, the packet driver should bind the physical index with the logical Ether object handle specified in *hEther*. This handle is used in receive indications to the stack.

llPacketClose *Close Driver and Unbind Logical Ether Object from Device ID*

Syntax void llPacketClose(uint dev);

Return Value None.

Description Closes the low level packet driver specified by the one's based index *dev*. The maximum value of *dev* is the number of devices returned from the *_llPacketInit()* function. After this call, the packet driver should no longer attempt to indicate received packets to the stack.

llPacketSetRxFilter *Set Packet Receive Filter*

Syntax void llPacketSetRxFilter(uint dev, uint filter);

Return Value None.

Description Called to set the types of packets that should be received via the receive indication function. Each level of filter is inclusive of the previous level. They are:

ETH_PKTFLT_NOTHING	No Packets
ETH_PKTFLT_DIRECT	Only directed Ethernet
ETH_PKTFLT_BROADCAST	Directed plus Ethernet Broadcast
ETH_PKTFLT_MULTICAST	Directed, Broadcast, and selected Ethernet Multicast
ETH_PKTFLT_ALLMULTICAST	Directed, Broadcast, and all Multicast
ETH_PKTFLT_ALL	All packets

llPacketGetMacAddr *Get MAC Address*

Syntax void llPacketGetMacAddr(uint dev, UINT8 *pbData);

Return Value None.

Description Copies the 6 byte MAC address of the physical device index *dev* into the supplied data buffer.

llPacketGetMCastMax *Get the Maximum Number of Multicast Addresses*

Syntax uint llPacketGetMCastMax(uint dev);

Return Value The maximum number of 6 byte MAC addresses that can be supplied for *llPacketSetMCast()*.

Description Called to get the maximum number of multicast addresses that can be supported on the physical packet device.

llPacketGetMCast *Get Multicast Address List*

Syntax uint llPacketGetMCast(uint dev, uint maxaddr, UINT8 *pbAddr);

Return Value The number of 6 byte MAC addresses written to *pbAddr*.

Description Called to get the current list of multicast addresses installed on the physical device. The maximum size of the list (supplied as an address count) is in *maxaddr*. The list is a contiguous stream of 6 byte addresses pointed to by *pbAddr*. The function returns the number of addresses in the list supplied.

llPacketSetMCast *Set Multicast Address List*

Syntax void llPacketSetMCast(uint dev, uint addrcnt, UINT8 *pbAddr);

Return Value None.

Description Called to install a list of multicast addresses on the physical device. The size of the list (supplied as an address count) is in *addrcnt*. The list is a contiguous stream of 6 byte addresses pointed to by *pbAddr*. The new list preempts any previously installed list, and thus an address count of ZERO removes all multicast addresses.

llPacketService *Service a Queued Packet*

Syntax void llPacketService();

Description This function is called to inform the driver that it may now indicate any queued packet buffers to the Ether object corresponding to the physical ingress device. Packet drivers must internally queue their own packets. Queued packets cause events to be sent to the scheduler that will in turn call this function.

Packets are passed to the Ether object via *EtherRxPacket()*.

llPacketSend *Send a Packet*

Syntax void llPacketSend(uint dev, PBM_Handle hPkt);

Description Called to send a packet out the physical packet device indicated by *dev*. The information about the packet (size and location) is contained in the PBM packet buffer specified by the handle *hPkt*. Once the packet has been sent, the packet buffer must be freed by calling *PBM_free()*.

The PBM packet buffer object is described in detail in Section A.3.

D.5 Low-Level Serial Port Driver (llSerial)

D.5.1 Synopsis

In the current directory structure, the serial port driver (llSerial) may or may not be part of the HAL directory (as it is an optional component). However, it is part of the HAL architecture, and should be programmed using the same guidelines used for the llTimer and llPacket drivers..

D.5.2 Function Overview

Application Functions:

_llSerialInit()	Initialize Driver Environment and Enumerate Devices
_llSerialShutdown()	Shutdown Driver Environment
_llSerialServiceCheck()	Check for packet activity
_llSerialSend()	Send Raw Data to the Serial Port

Kernel Layer Functions:

llSerialOpen()	Open Driver in Character Mode
llSerialClose()	Close Driver Character mode
llSerialOpenHDLC()	Open Driver HDLC Session
llSerialCloseHDLC()	Close Driver HDLC Session
llSerialConfig()	Set Serial Port Configuration

llSerialHDLCPeerMap()	Update the HDLC encoding peer CMAP
llSerialService()	Service HDLC Packets
llSerialSendPkt()	Send a Serial Data Packet

D.5.3 Low-Level Serial API Functions

The low level support layer must provide the following functions:

_llSerialInit *Initialize Driver Environment and Enumerate Devices*

Syntax uint _llSerialInit(STKEVENT_Handle hEvent);

Return Value Returns the number of physical serial devices.

Description This function is called by NETCTRL to initialize the system to use the serial port. It also enumerates all the physical devices in the system, and returns a device count. The stack will then call the *llSerialOpen()* function and/or the *llSerialOpenHDLC()* function for each physical device it requires.

The *hEvent* calling parameter is a handle to a STKEVENT object that must be signaled whenever a serial packet (or raw data) is received. This STKEVENT object is discussed in Section A.4.

_llSerialShutdown *Shutdown Driver Environment*

Syntax void _llSerialShutdown();

Return Value None.

Description This function is called by NETCTRL to indicate a final shutdown of the serial driver environment. When called, there should be no currently open serial drivers, and *_llSerialInit()* will be called again before any call to *llSerialOpen()* or *llSerialOpenHDLC()*.

_llSerialServiceCheck *Check for Serial Port Activity*

Syntax uint _llSerialServiceCheck(uint fTimerTick);

Return Value None.

Description This function is called by NETCTRL to check if serial packets (or data) are available from the serial device. In a polling system, this function is called continuously. In an interrupt driven semaphore system, it is called when packet activity is indicated via the STKEVENT object, and also by the scheduler at 100mS timer intervals for dead man polling checks.

In both polling and interrupt environments, the *fTimerTick* flag will be set whenever a 100mS timer tick has occurred.

If any *new* serial packets are detected from within this function, the packet driver should signal the STKEVENT object in the passive mode (do not set the *fHwAsynch* flag in the *STKEVENT_signal()* function). This only applies to new packet events detected from within this function. The STKEVENT object is discussed in Section A.4.

Finally, if the driver is only open in character mode (not HDLC), and there are characters for the character mode device waiting, they are passed into the user application from this function by calling character mode input callback function passed to *llSerialOpen()*.

_llSerialSend — *Send Raw Data to the Serial Port*

Syntax

uint _llSerialSend(uint dev, UINT8 *pBuf, uint len);

Return Value

The number of bytes sent to the serial port.

Description

This function is called by the application to send raw unpacketized serial data to the serial port. This function may only be called when the serial driver is not open for HDLC mode. The function returns the number of bytes sent, which will always be either the number of bytes it was told to send specified by the *len* parameter, or NULL on an error.

Note that this function is provided mainly for convenience to the application programmer. The implementation of this function is to packetize the data specified in the *pBuf* and *len* parameters into a PBM buffer, and then call *SerialSendPkt()*.

llSerialOpen — *Open Driver in Character Mode*

Syntax

uint llSerialOpenCharmode(uint dev, void (*pCharmodeRxCb)(char c));

Return Value

This function should return 1 on success, and 0 on failure.

Description

Opens the low level serial driver specified by the one's based index *dev* in character mode. The maximum value of *dev* is the number of devices returned from the *_llSerialInit()* function.

Character mode input simply passes all characters received at the port to the character mode receiver.

When opening the device, the driver should save the callback function pointer *pCharmodeRxCb*. This function is called for each character received while in character mode when the *_llSerialServiceCheck()* function is called. Serial drivers queue up serial data, signaling an event to the STKEVENT object passed to *_llSerialInit()*, and then pass the serial data to the application callback function from within the *_llSerialServiceCheck()* function.

When the driver is opened in HDLC mode, no character mode input is received. When the HDLC mode is closed, the character mode becomes active again.

llSerialClose — *Close Driver Character Mode*

Syntax

void llSerialClose(uint dev);

Return Value

None.

Description

Closes the character mode of the low level serial driver specified by the one's based index *dev*. Once called, the serial driver should not attempt to call any character mode callback function.

llSerialOpenHDLC *Open Driver HDLC Session*

Syntax

uint llSerialOpenHDLC(uint dev, HANDLE hHDLC, void (*cbTimer)(HANDLE h), void (*cbHDLCInput)(PBM_Handle hPkt));

Return Value

This function should return 1 on success, and 0 on failure.

Description

Opens the low level serial driver specified by the one's based index *dev* in HDLC mode. The maximum value of *dev* is the number of devices returned from the *_llSerialInit()* function.

The *hHDLC* parameter is a handle to the HDLC device. Any HDLC packet received has its Rx interface in the PBM packet buffer set to this device handle.

The callback function *cbTimer* is called by the driver every second to drive any timeouts required by the caller. Note the serial driver calls *cbTimer* from kernel mode.

Serial drivers queue up HDLC packets. When a complete HDLC packet is ready, the driver signals an event to the STKEVENT object passed to *_llSerialInit()*, and then passes the HDLC packet (as a PBM packet buffer) to the application callback function *cbHDLCInput* from within the *llSerialService()* function.

This is similar to character mode operation, but different because the entire packet is passed over at one time, and it is done from the *llSerialService()* function, not from *_llSerialServiceCheck()* as with character mode data. The *cbHDLCInput* function is called from kernel mode while the character mode application callback is not.

When the driver is in HDLC mode, the driver receives serial data as HDLC packets, and creates a PBM packet buffer object to hold each HDLC frame. Note that the HDLC flag character (0x7E) is always removed from the HDLC packets. The HDLC packet passed to the *cbHDLCInput* function is formatted as follows:

Addr (FF)	Control (03)	Protocol	Payload	CRC
1	1	2	1500	2

The serial driver processes the HDLC packet data as it arrives to remove any escaped characters and to verify the CRC. When a HDLC packet is ready, the driver signals an event to the STKEVENT object.

llSerialCloseHDLC *Close Driver HDLC Session*

Syntax void llSerialCloseHDLC(uint dev);

Return Value None.

Description Closes the HDLC mode of the low level serial driver specified by the one's based index *dev*. Once called, the serial driver should not attempt to indicate HDLC frame buffers to the scheduler or stack. Any queued buffers should be flushed.

llSerialConfig *Configure Serial Port*

Syntax void llSerialConfig(uint dev, uint baud, uint mode, uint flowctrl);

Return Value None.

Description This function is called to configure the serial port attributes for the indicated device.

The value of *baud* is the baud rate, and must be an even denominator of 230400, up to a maximum baud rate of 230400. For example: 230400, 115200, 57600, 38400, 28800, and 19200 are all legal values, while 56000 is not.

The value of *mode* indicates the mode of the device including data bits, parity, and stop bits. Only the two most commonly used modes are defined:

HAL_SERIAL_MODE_8N1	8 Data Bits, No Parity, 1 Stop Bit
HAL_SERIAL_MODE_7E1	7 Data Bits, Even Parity, 1 Stop Bit

The value of flowctrl indicates the desired flow control operation. Legal values for this parameter are:

HAL_SERIAL_FLOWCTRL_NONE	No Flow Control
HAL_SERIAL_FLOWCTRL_HARDWARE	Hardware Flow Control

This function can be called before or after the device is opened.

llSerialHDLCPeerMap *Update the HDLC Encoding Peer CMAP*

Syntax void llSerialHDLCPeerMap(uint dev, UINT32 peerMap);

Return Value None.

Description When in HDLC mode, the serial driver sends all serial data as HDLC frames. This requires it to add the frame flag characters, and do any character escaping necessary to encode the frame for transmission over the serial link. This includes escaping characters that appear in the peer's character map (CMAP).

By default, the CMAP is set to 0xFFFFFFFF. For character codes 0 to 31, if the bit (1<<charval) is set in the CMAP, then the serial driver performs an HDLC escape sequence when sending the character in an HDLC frame.

This function allows the application to update the peer's CMAP as it gets information from the peer allowing it to do so.

llSerialService — *Service HDLC Packets*

Syntax

void llSerialService();

Return Value

None.

Description

This function is called to inform the driver that it may now indicate any queued HDLC buffers to the HDLC callback function corresponding to the serial port. Serial drivers internally queue a PBM packet buffer for each HDLC frame received. When a new packet is received, the driver signals the STKEVENT object, which will cause this function to be called by the network scheduler.

llSerialSendPkt — *Send a Serial Data Packet*

Syntax

void llSerialSendPkt(uint dev, PBM_Handle hPkt);

Return Value

None.

Description

Called to send a serial data packet out the physical serial device indicated by *dev*. The information about the packet (size and location) is contained in the PBM packet buffer specified by the handle *hPkt*. Once the packet has been sent, the packet buffer must be freed by calling *PBM_free()*.

The data is treated as raw bytes when the driver is not open in HDLC mode. When in HDLC mode, the data packet is an HDLC frame with the following format:

Addr (FF)	Control (03)	Protocol	Payload	CRC
1	1	2	1500	2

Note that the CRC on the packet does not need to be valid. The serial port driver will validate the CRC when the packet is sent. However, the 2 byte space-holder for the CRC must be present in the packet.

Web Programming with the HTTP Server

The easiest way to get information from an embedded network device is through the web server. The HTTP server pulls files from the embedded file system (EFS) that is included in the NDK software package's OS adaptation layer. These files can be compiled into the DSP application, located on a network file system, a memory-based file system, or on a physical disk interfaced to the DSP. The NDK HTTP server accesses files through the EFS application interface, which can be ported to any file system desired. The server currently supports the HTTP/1.0 protocol.

Common Gateway Interface (CGI) programs execute on a web server and process input from a user. They are useful as interfaces to services running on the device. Writing CGI programs for the NDK is relatively simple and only requires a few specific functions. A single CGI interface function can be written to support both HTTP POST requests and GET requests.

The CGI program is built from a single C callable entry-point (or CGI function). Each CGI function is called on its own independent task thread. The task threads are created with a priority of OS_TASKPRINORM and a stack size of OS_TASKSTKHIGH. Note that consecutive calls to the same CGI function will not be on the same task thread. Thus, CGI functions cannot share sockets from one call to the next. In general, there is no persistent data in a CGI function.

Also, file detection of CGI functions is done purely on the file extension. If the file ends with .cfg (case insensitive), then a POST or a GET of the file will result in a call to the CGI function mapped to that filename. A POST call to a non-CGI file is not allowed.

Topic **Page**

E.1 Adding Web Content

E.1.1 Operation

As previously mentioned, the HTTP server allows access to files using the embedded file system (EFS) API. The default installation of this API is a RAM based file system that resides in the OS adaptation layer. This OS adaptation layer allows the HTTP server to work on any file storage device contained in the system.

The default RAM based file system is built up mainly from a standard file I/O API, with the addition of some private functions. These private functions allow files to be created and destroyed by passing in memory pointers to where they are stored. These functions are fully documented in Section 2.6.

E.1.2 Converting Standard HTML Files

The example code supplied with the NDK adds Web pages by converting them from binary HTML files into data arrays declared in C. An MS-DOS utility **binsrc** is supplied to allow conversion of files to a C array.

The calling format for binsrc is:

binsrc <*input file name*> <*output file name*> <*identifier*>

Parameters:

input file name	File to be converted
output file name	Name for file containing C data representation of the input file name
identifier	C name for data

For example, to convert an HTML file default.htm for use by EFS, the following command could be executed from the Windows command window:

```
binsrc default.htm default.c DEFAULT
```

The file default.c would contain the following:

```
#define DEFAULT_SIZE 1610
unsigned char DEFAULT[] = {0x3C, 0x21, 0x64, 0x6F, 0x63, 0x74, 0x79, 0x70, 0x65, 0x20, 0x68,
0x74, ...
```

E.1.3 Declaring HTML Files to EFS

Once the HTML file is converted to a memory image, the file is declared to the EFS file system by calling the function *efs_createfile()*. All the HTML files are typically created at the same time, during initialization, and before the HTTP server is actually invoked. In the example code, there are two functions used, *AddWebFiles()* and *RemoveWebFiles()*. These functions include all the code necessary to initialize and clean up the EFS file environment.

An example implementation of *AddWebFiles()* is shown below. Note the addition of two file creation calls. The first call to *efs_createfile()* creates the file declared in default.c as converted from the file default.htm. The second call creates a CGI file that is a C function entry-point. When a post is attempted to `sample.cgi`, the function *cgiSample()* is called.

```
// Include our externally converted pages
#pragma DATA_SECTION(DEFAULT, "HTMLDATA");
#include "default.c"

// Declare our CGI function entry point
static int cgiSample(int htmlSock, int ContentLength);

// Our function to initialize EFS with our Web files
```

```
void AddWebFiles(void)
{
    efs_createfile("index.html", DEFAULT_SIZE, DEFAULT);
    efs_createfile("sample.cgi", 0, (UINT8 *)cgiSample);
}
```

Once the above code is run, the EFS system is ready for the HTTP server to serve up the content. Note the inclusion of the #pragma to place the converted Web page into a memory section named HTMLDATA. This allows the page to be placed out of the way by specifying the section's location in the linker command file.

E.1.4 Cleaning up HTML Files

Because the EFS system uses memory records to simulate file content from static data, the system should be flushed or cleaned when shutting down or rebooting. In the example code, the function *RemoveWebFiles()* is called when the EFS files are no longer required.

An implementation of *RemoveWebFiles()* that corresponds to the *AddWebFiles()* function shown above would be as follows:

```
void RemoveWebFiles(void)
{
    efs_destroyfile("sample.cgi");
    efs_destroyfile("index.html");
}
```

E.2 Writing CGI Functions

E.2.1 Adding Functions to the EFS

CGI programs must be in the EFS for the HTTP server to see them. An example of this was shown in the previous section by adding an entry for the file sample.cgi that translated into the C function *cgiSample()*. Whenever a POST is made to the file `sample.cgi`, the *cgiSample()* function is called.

E.2.2 CGI Function Declaration

The standard declaration for a CGI function in C is:

Function ***CGI Function Declaration***

Syntax static int cgiSample(int htmlSock, int ContentLength, char *pArgs);

Parameters

htmlSock	The network socket on which the HTTP POST was issued
ContentLength	The size of the POST content waiting on socket *htmlSock*
pArgs	Pointer to NULL terminated arguments from a CGI 'GET'

Return Value All CGI functions return 1 if the input socket is left open, and 0 if it is closed or transferred to another thread.

Description This function reads in the HTTP POST content from the socket *htmlSock*, and writes out an HTML reply based on the function and the form content read. The size of the form content is specified by *ContentLength*.

The CGI function must decide whether or not to close the socket on which the POST arrived. By default, the socket is normally left open, but in some cases may need to be closed. It is also possible that the CGI function may wish to take control of the socket and close it at a later point in time. Note in this latter case, the socket would be transferred to another thread, using the *fdGetFileHandle()* and *FileHandleGetFd()* function calls.

The function must return either 0 or 1 to indicate the status of the socket *htmlSock*. If the socket is closed or passed on to another task, the function returns 0. If the socket is still active, the function returns 1.

When there is any doubt whether or not to close the socket, the socket is typically left open for the HTTP server to close when appropriate.

The *ContentLength* argument is the size of the CGI argument still to be read from the socket. On a CGI GET operation, the arguments have already been read from the socket and are passed as a NULL terminated string in the *pArgs* parameter. Note that in any given CGI call, either one or both of these parameters can be NULL. It is also possible for *pArgs* to point to a zero length string.

E.2.3 Parsing CGI Form Data

The first task a CGI function will most likely perform is to read the POST form data from the socket. This can be done easily because two of the calling arguments to the function are the socket to read and the size of the data. To remain reentrant, the CGI function should allocate its memory buffer to hold the form data.

After reading in the data from the socket, each form entry can be parsed from the from by using the supplied example function: *cgiParseVars()*.This function can be used to parse the NULL terminated option string that may also be passed to the CGI function. The formal definition of the function is shown below.

Note that this function replaces *parsePostVars()*,a similar function that was included in earlier versions of the NDK. The *parsePostVars()* function was not reentrant, and has been purged from the NDK release. The source code to *cgiParseVars()* is included in the example application code shipped with the stack.

cgiParseVars ***Parse CGI Form POST Input***

Syntax

char *cgiParseVars(char PostInput[], int *pParseIndex);

Parameters

PostInput[]	Pointer to the form data read in from the HTTP request socket
pParseIndex	Pointer to an int holding the current parse position (initially zero)

Return Value

A pointer to a NULL terminated string within *PostInput[]*, signifying the name or value of a form entry. Also updates the value pointed to by *pParseIndex*.

Description

Reads input from a CGI POST operation pointed to by *PostInput[]* at an offset pointed to by *pParseIndex* and returns in sequence a pointer to the name and then the value of each post entry. This function modifies the data in *PostInput[]*. It also updates the current parsing position in the variable *pParseIndex*. The parse index must be set to 0 on initial call.

On the initial call to this function, the integer value pointed to by *pParseIndex* should contain zero.

On reaching the end of the input, the function sets *pParseIndex* to -1. If called again, the function will return a NULL pointer and leave the value of *pParseIndex* unchanged.

E.2.4 Parsing CGI Multi-Part Form Data

In some cases, it is preferable to use a multi-part form when posting CGI data. The multi-part form is specified in the HTML code by adding the tag **ENCTYPE="multipart/form-data"** to the form type. When this form type is used, form entries are sent in a slightly different format than with the standard form, thus an alternate CGI parsing function is required.

After reading in the data from the socket, each form entry can be parsed from the multi-part form by using the supplied example function: *cgiParseMultiVars()*.This function parses the post data into individual records. The formal definition of the function is shown below.

cgiParseMultiVars ***Parse CGI Form Multi-Part POST Input***

Syntax

int cgiParseMultiVars(char *buffer, int buflen, CGIPARSEREC *recs, int maxrecs);

Parameters

buffer	Buffer holding the entire post content
buflen	Length of the post content
recs	Pointer to an array of records of type CGIPARSEREC
maxrecs	The maximum number of records that can be written to *recs*

Return Value

The number of valid records parsed, or -1 on a parsing error.

Description

Reads input from a CGI POST operation pointed to by *buffer*, with length *buflen*, and returns a collection of CGIPARSEREC records to *recs*. The caller must provide buffer space to hold *recs*, and indicate the maximum number of records that can be written to the buffer in *maxrecs*.

The CGIPARSEREC record is defined as follows:

```
typedef struct {
    char *Name;     // NULL terminated entry "name"
    char *Filename; // NULL terminated "filename" or NULL if not a file
    char *Type;     // NULL terminated "Content-Type" or NULL if no type
    char *Data;     // Pointer to file or entry data (NULL term for string)
    int DataSize;   // Length of data (valid on strings and file data)
} CGIPARSEREC;
```

This function modifies the data in *buffer* to add string delimiters. This function should only be called once to parse all entries from the form data.

E.2.5 Sending HTTP/HTML Replies

After parsing the CGI POST form data, the CGI function should send some sort of reply to the requesting client. The reply takes the form of an HTTP message signifying success or error, potentially followed by HTML data.

The HTTP server supplies several functions to aid in building and sending HTTP data over the socket. In addition, the example applications contain various MACROS than can also help in initially developing a CGI function. The HTTP functions are fully described at the end of this section, but the main reply functions are usually one of the following:

httpSendFullResponse()	Send full HTTP response, including a status code and an HTML file
httpSendErrorResponse()	Send full HTTP error response, including an HTML message

or

httpSendStatusLine()	Send HTTP status response, including a status code and content type
httpSendEntityLength()	Send HTTP *content length* and terminate HTTP header (optional)
httpSendClientStr()	Used after *httpSendStatusLine()* to send content data

As an example of using these functions, consider the two response MACROS included in inc\nettools\httpif.h.

```
//
// Common error responses
//
#define http404(Sock) httpSendErrorResponse(Sock, HTTP_NOT_FOUND)
#define http501(Sock) httpSendErrorResponse(Sock, HTTP_NOT_IMPLEMENTED)
```

These MACROS use the error response function to send a full error message to the client. Alternately, the *httpSendStatusLine()* function can be used to start a message that is completed by the application. Under normal circumstances, a CGI function will use the *httpSendStatusLine()* function to send an OK message to the client, followed by the *httpSendClientStr()* function to send client data in the form of a NULL terminated string. Note that an additional carriage return and line feed are required to separate the header from the HTML data.

For example, the following code sends a quick Success message.

```
// Send response status
httpSendStatusLine(Sock, HTTP_OK, CONTENT_TYPE_HTML);

// Terminate the response header
httpSendClientStr(Sock, CRLF);

// Send the Success Message
httpSendClientStr(Sock, "<html><h1>Success!</h1><br></html>");
```

Note that the *httpSendClientStr()* function replaces the *httpSendClientData()*function from earlier releases of the NDK. For data sizes that can be represented by an integer, client data can also be sent simply by calling the sockets *send()* function.

E.2.6 HTML Error Response

The HTTP server generates a generic error response message for several possible HTTP errors. The function *httpSendErrorResponse()* is part of this function. The error response consists of two parts, the HTTP header and the HTML response message. It is the HTML message that is displayed to the web browser when an error occurs.

The default HTML message used by the HTTP server is quite plain. For example, on the error 404, it generates:

<html><body><h1>HTTP/1.0 404 - File Not Found</h1></body></html>

Some application developers may wish to enhance the HTML generation of errors. This is done by hooking a callback function into the HTTP server error processing. The callback hook is defined as:

_extern int (*httpErrorResponseHook)(SOCKET Sock, int StatusCode);

Any function using the callback must generate the content length tag, and then the entire HTML response page. (The content length is the length of the HTML response.) It can be written using the *httpSendEntityLength()* function.

If the application does not wish to handle the error, it can return NULL indicating that it did not handle the error. In this case, the HTTP server will use the default HTML. If the application returns 1, this tells the HTTP server that the HTTP response was completed by the callback function.

The *httpErrorResponseHook* function pointer is NULL by default. If an application needs to install a callback to this pointer, the value should be set before the HTTP server is initialized.

As an example of how the callback function may look, here is the default error response function. Any substitute function provided by the application would be quite similar:

```
typedef struct _codestr {
    int code;
    char *string;
} CODESTR;


// Note MAX string length is 30 (since Data[] is 80 bytes)
```

```
CODESTR codestr[] = {
    { HTTP_OK, " OK" },
    { HTTP_NO_CONTENT, " No Content" },
    { HTTP_AUTH_REQUIRED, " Authorization Required" },
    { HTTP_NOT_FOUND, " File Not Found" },
    { HTTP_NOT_IMPLEMENTED, " Not Implemented" },
    { HTTP_NOT_ALLOWED, " Not Allowed" },
    { 0, " Unknown" } };

int httpSendErrorHTML(SOCKET Sock, int StatusCode)
{
    char Data[80];
    int I;

    // Build the HTML response into Data[]
    sprintf(Data, "<html><body><h1>HTTP/1.0 %3d -", StatusCode);
    for(i=0;codestr[i].code && codestr[i].code!=StatusCode;i++);
    strcat(Data, codestr[i].string);
    strcat(Data,"</h1></body></html>");

    // Send the length of the HTML response
    // (this also terminates the HTTP header)
    httpSendEntityLength(Sock, strlen(Data));

    // Send the respone data
    httpSendClientStr(Sock, Data);
    return(1);
}
```

E.3 HTTP Authentication

The HTTP server included in the NDK supports the Basic method of HTTP authentication, which is MIME encoding of the username and password.

As with other HTTP functionality, the HTTP server calls an EFS function to perform file access authentication. The EFS function used is *efs_filecheck()*. The function is passed the filename of the file to be authenticated, and the username and password of the user attempting to access the file.

The exact method used to designate a file as protected and to authorize individual access, is determined by the implementation of the *efs_filecheck()* function. This section describes the operation of the example implementation of *efs_filecheck()* provided in the NDK.

E.3.1 Authorization Realms

Regardless of implementation of the authentication scheme at the EFS layer, the HTTP server understands the authority system to be based on four authorization realms. The realms are enumerated 1 to 4, and the authorization realm index (when required) is returned to the HTTP server by the *efs_filecheck()* function.

When the HTTP server indicates to the client that authorization is required, it supplies the name of the authorization realm to the client. The application programmer can specify the name of each authorization realm by using the configuration system. The configuration tag `CFGTAG_SYSINFO` is used for storing authorization realm names. The item numbers used for the four realms are `CFGITEM_SYSINFO_REALM1` through `CFGITEM_SYSINFO_REALM4`.

For example, to set the name of authorization realm 1, while building the configuration, the programmer could write:

```
// Name our authentication group for HTTP (Max size = 31)
// This is the authorization "realm" name returned by the HTTP
// server when authentication is required on group 1.
CfgAddEntry( hCfg, CFGTAG_SYSINFO, CFGITEM_SYSINFO_REALM1,
             0, 30, (UINT8 *)"DSP_CLIENT_DEMO_AUTHENTICATE1", 0 );
```

If no realm name is supplied in the configuration, then a default realm name is used by the HTTP server.

E.3.2 User Accounts

How and whether user accounts are stored in the system is entirely up to the system programmer. The user account is only accessed directly in the *efs_filecheck()* function.

However, the default implementation of *efs_filecheck()* uses the configuration system to access usernames and passwords. These user accounts can be added to the configuration system at any time. As an example, the following code adds a sample account to access authorization realm 1. The username and password are simply *username* and *password* respectively:

```
CI_ACCT CA;

// Create a sample user account who is a member of realm 1.
strcpy( CA.Username, "username" );
strcpy( CA.Password, "password" );
CA.Flags = CFG_ACCTFLG_CH1;        // Make a member of realm 1

rc = CfgAddEntry( hCfg, CFGTAG_ACCT, CFGITEM_ACCT_REALM,
                  0, sizeof(CI_ACCT), (UINT8 *)&CA, 0 );
```

E.3.3 Designating Protected Files

As with the authorization user accounts, the method of how a file is designated as protected depends on the implementation of the *efs_filecheck()* function.

In the default implementation, files are grouped for authorization by their first level directory. For example, the files index.html and banner.gif would both carry the same authorization requirements, while the files mydir/sample.cgi and mydir/sample.htm would carry a different authorization. The file group is marked for authorization by placing a special file in the directory, named %R%. This file is exactly 4 bytes long, and contains an integer value, being the realm index 1 to 4. If there is no %R% file in the directory, then no authorization is required.

For example, the following code sets up a small web page in an unprotected space (the root directory), and then sets up sample.cgi and sample.htm in a protected directory, requiring authentication on authorization realm 1.

```
//
// The authentication scheme works by looking for files
// named %R% in the subdirectory of any given filename, or in
// the root directory if no subdirectory exits. The file
// contains a single 4 byte int that is the authentication
// realm index. If there is no file, there is no authentication.
//
// Note for this implementation, only the first subdirectory level
// is validated.
//
// The int "OurRealm" will be our "%R%" realm file, forcing any file
// located in the same directory to be authenticated on realm 1. The
// system supports realms 1 to 4.
//
// Note that we are only going to protect the "protected/" subdir,
// but it is also possible to protect the entire web site by putting
// a %R% file in the root. Also, you can have the root protected
// on (say) realm 1, and a subdir on (say) realm 2, allowing for
// "users" (members of realm 1) and "superusers" (members of both
// realm 1 and realm 2).
//
static int OurRealm = 1;

void AddWebFiles(void)
{
    efs_createfile("index.html", DEFAULT_SIZE, DEFAULT);
    efs_createfile("logobar.gif", LOGOBAR_SIZE, LOGOBAR);
    efs_createfile("dspchip.gif", DSPCHIP_SIZE, DSPCHIP);
    efs_createfile("inform.cgi", 0, (UINT8 *)cgiInform);
    efs_createfile("protected/%R%", 4, (UINT8 *)&OurRealm);
    efs_createfile("protected/sample.htm", SAMPLE_SIZE, SAMPLE);
```

```
    efs_createfile("protected/sample.cgi", 0, (UINT8*)cgiSample);

}
```

E.4 CGI Function Example

As an example of using all the concepts described so far, consider a simple example. Assume an applications programmer wishes to create a Web form that inputs and processes user data.

E.4.1 Create the HTML Page

The HTML page can be created with an HTML editor, or by hand. For this example, there is an HTML page that contains a simple CGI form. The contents of the example page, `default.htm` are shown below.

```
<html>
<head><title>CGI Sample</title></head>
<body>
  <h1>CGI Sample Form</h1>
  <hr WIDTH="100%">

  Fill in the form fields and hit 'Submit'.
  <form name="my_form" method="POST" action="sample.cgi">
    Name: <input type="text" name="name"><br>
    I dislike spam: <input type="checkbox" name="spam" value="no!"><br>
    Favorite Pizza:
      <input type="radio" name="pizza" value="pepperoni"> Pepperoni
      <input type="radio" name="pizza" value="sausage"> Sausage
      <input type="radio" name="pizza" value="cheese" checked> Cheese
      <input type="radio" name="pizza" value="other"> Other
      <br>
    Favorite Color: <select name="color">
      <option value="red"> Red
      <option value="green"> Green
      <option value="blue"> Blue
      <option value="yellow"> Yellow
      <option value="cyan"> Cyan
      <option value="magenta"> Magenta
      <option value="black"> Black
      <option value="white"> White
      </select>
    </p>
    <input type="submit"> <input type="reset">

  </form>
</body>
</html>
```

The next step performed is to convert this HTML file to C source file, as seen in Section E.1.2. Once the page is in C source code form, it can be added to the program.

E.4.2 Create the Base WEBPAGE Source File

Once the HTML pages are ready in source form, the main WEBPAGE.C source file is created. This file will perform all the necessary Web processing in the example. The basic source code declares the HTML pages as files to the EFS file system. To do this, it exports two functions called from the main network initialization routine, *AddWebFiles()* and *RemoveWebFiles()*. Note that a CGI function is also declared to handle processing of the CGI form contained on the Web page, called **sample.cgi**.

The source code as defined so far is shown below.

```
static int cgiSample(int htmlSock, int ContentLength, char *pArgs )
{
    char    *name = 0, *spam = 0, *pizza = 0, *color = 0;
    char    *buffer, *key, *value;
    int     len;
    int     parseIndex;
    char    htmlbuf[MAX_RESPONSE_SIZE];

    // The pArgs parameter is used for passing arguments
    // on the address line using the '?' operator. It is
    // typcially not used on a CGI POST

    // 1. Read in the request data

    // First, allocate a buffer for the request
    buffer = (char*) mmBulkAlloc( ContentLength + 1 );
    if ( !buffer )
        goto ERROR;

    // Now read the data from the client
    len = recv( htmlSock, buffer, ContentLength, MSG_WAITALL );
    if ( len < 1 )
        goto ERROR;


    // 2. Parse request using cgiParseVars(), or a similar function

    // Setup to parse the post data
    parseIndex = 0;
    buffer[ContentLength] = '\0';

    // Process request variables until there are none left to do
    {
        key   = cgiParseVars( buffer, &parseIndex );
        value = cgiParseVars( buffer, &parseIndex );

        if( !strcmp("name", key) )
            name = value;
        else if( !strcmp("pizza", key) )
            pizza = value;
        else if( !strcmp("spam", key) )
            spam = value;
        else if( !strcmp("color", key) )
            color = value;
    } while ( parseIndex != -1 );


    // 3. Process request in some meaningful way . . .
    // (OK, we really don't do this here.)

    // 4. Send a response. Keep in mind the first line of the
    //    response should indicate whether the request was
    //    successful or not.

    httpSendStatusLine(htmlSock, HTTP_OK, CONTENT_TYPE_HTML);

    // 5. Send appropriate headers

    // No more header data to send - CRLF terminates header
    html( CRLF );

    // 6. Send the response data

    // Build our HTML response
```

```
    // Here we'll just echo back the input we received
    // to an HTML table.
    //
    html("<html><body text=#000000 bgcolor=#ffffff>\r\n");
    html("<h1>Form Information</h1>");
    html(divider);
    html("<table border cellspacing=0 cellpadding=5>\r\n");

    if( name )
    {
      sprintf( htmlbuf, tablefmt, "Name:", name );
      html( htmlbuf );
    }

    if( spam )
    {
      sprintf( htmlbuf, tablefmt, "Likes Spam:", spam );
      html( htmlbuf );
    }

    if( pizza )
    {
      sprintf( htmlbuf, tablefmt, "Favorite Pizza:", pizza );
      html( htmlbuf );
    }

    if( color )
    {
      sprintf( htmlbuf, tablefmt, "Favorite Color:", color );
      html( htmlbuf );
    }

    html("</table><br>\r\n");
    html(divider);
    html("<a href=index.html>Return to Main Page</a><br><br>\r\n");
    html("</body></html>\r\n");

ERROR:
    if( buffer )
        mmBulkFree( buffer );

    return( 1 );
}
```

E.5 HTTP Server Exported Functions

The HTTP server module exports several functions and strings to aid in the creation of a CGI function. This section contains the formal specification for these functions. The first part of this appendix describes how to use these functions in creating a HTTP CGI function in C.

E.5.1 Commonly Used Strings

To aid in the creation of the response data, some commonly used HTML strings can be defined. Some of these are already defined in the `HTTPIF.H` file. These include the following (note that all entries, except the first three, include a trailing space character.):

Global Name	String Value
`DEFAULT_NAME`	`"index.html"`
`CRLF`	`"\r\n"`
`SPACE`	`" "`
`HTTP_VER`	`"HTTP/1.0 "`
`CONTENT_LENGTH`	`"Content-length: "`
`CONTENT_TYPE`	`"Content-type: "`
`CONTENT_TYPE_APPLET`	`"application/octet-stream "`
`CONTENT_TYPE_AU`	`"audio/au"`
`CONTENT_TYPE_DOC`	`"application/msword "`
`CONTENT_TYPE_GIF`	`"image/gif "`
`CONTENT_TYPE_HTML`	`"text/html "`
`CONTENT_TYPE_JPG`	`"image/jpeg "`
`CONTENT_TYPE_MPEG`	`"video/mpeg "`
`CONTENT_TYPE_PDF`	`"application/pdf "`
`CONTENT_TYPE_WAV`	`"audio/wav "`
`CONTENT_TYPE_ZIP`	`"application/zip "`

E.5.2 Function Overview

The basic HTTP Server exported functions are as follows:

httpSendStatusLine()	Send the status of this request to the client
httpSendClientStr()	Send NULL terminated string data to client
httpSendFullResponse()	Send a full formatted response to the client
httpSendEntityLength()	Send the content length and terminate HTTP header
httpSendErrorResponse()	Send a full formatted response to the client

E.5.3 HTTP Server Exported API Functions

httpSendStatusLine *Send the Status of this Request to the Client*

Syntax

void httpSendStatusLine(int Sock, int StatusCode, char *ContentType);

Parameters

Sock	Socket on which to send
StatusCode	HTTP status code of the request
ContentType	HTTP type string of the response

Return Value

None.

Description

Sends a formatted response message to *Sock* with the given status code and content type. The value of *ContentType* can be NULL if no ContentType is required.

The status code and content type should match HTTP standard definitions. Some content type strings are listed in Section E.5.1. The pre-defined status codes include:

HTTP_OK	(200)
HTTP_NO_CONTENT	(204)
HTTP_AUTH_REQUIRED	(401)
HTTP_NOT_FOUND	(404)
HTTP_NOT_ALLOWED	(405)
HTTP_NOT_IMPLEMENTED	(501)

httpSendClientStr *Send NULL Terminated String Data to Client*

Syntax

void httpSendClientStr(int Sock, char *Response);

Parameters

Sock	Socket on which to send
Response	Pointer to NULL terminated string

Return Value

None.

Description

This function sends the indicated NULL terminated response string to the indicated client socket. In other words, it calls *strlen()* and *send()*.

httpSendFullResponse *Send a Full Formatted Response to the Client*

Syntax

void httpSendFullResponse(int Sock, int StatusCode, char *RequestedFile);

Parameters

Sock	Socket on which to send
StatusCode	HTTP status code of the request
RequestedFile	Pointer to filename of file to include in body

Return Value

None.

Description

Sends a full formatted response message to *Sock*, including the file indicated by the filename pointed to by *RequestedFile*. The status code for this call is usually HTTP_OK.

httpEntityLength *Send the Content Length and Terminate HTTP Header*

Syntax

void httpSendEntityLength(SOCKET Sock, INT32 EntityLength);

Parameters

Sock	Socket on which to send
EntityLength	Length of the entity (usually HTML page) to follow the HTTP header

Return Value

None.

Description

Writes out the entity length tag, and terminates the HTTP header with an additional CRLF. Because the header is terminated, this must be the last tag written. Entity data should follow immediately.

httpSendErrorResponse *Send a Full Formatted Error Response to the Client*

Syntax

void httpSendErrorResponse(int Sock, int StatusCode);

Parameters

Sock	Socket on which to send
StatusCode	HTTP status code of the request

Return Value

None.

Description

Sends a full formatted error response message to *Sock*, including a small HTML file displaying the status code. For example, HTTP_NOT_FOUND would generate:

<html><body><h1>HTTP/1.0 404 – File Not Found</h1></body></html>