Porting TCP/IP

Version 1.6

Programmer's Guide

ARM

ARM DUI 0144B

Porting TCP/IP
Programmer's Guide

Release Information

The following changes have been made to this document.

Change History

Date	Issue	Change
Sept 2000	A	First release of independent TCP/IP version (DUI 0144), without PPP information.
June 2001	B	Second release. Minor changes.

Proprietary Notice

This document is intended only to assist the reader in the use of the product. ARM Ltd shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Contents
TCP/IP Programmer's Guide

Preface

This preface introduces the ARM TCP/IP implementation and its documentation. It contains the following sections:

About this book

This guide is provided with the ARM Portable TCP/IP stack sources.

It is assumed that the ARM TCP/IP sources are available as a reference. It is also assumed that the reader has access to a C language programmer's guide and the *ARM Architecture Reference Manual*.

Intended audience

This Programmer's Guide is written for a moderately-experienced C programmer, with a general understanding of TCP/IP, who wants to port the stack to a new environment.

Using this book

This guide is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter to learn about porting in general and the system requirements for using the TCP/IP protocol stack source.

Chapter 3 *TCP/IP Porting*

Read this chapter for step-by-step instructions on porting and testing.

Chapter 2 *TCP/IP API Functions*

Read this chapter for a description of the user-provided functions required for porting the ARM TCP/IP stack.

Chapter 3 *DHCP Client Functions*

Read this chapter for a description of the DHCP function calls used to request information for an interface.

Chapter 4 *Sockets*

Read this chapter for an introduction to sockets, and for a description of the sockets API.

Chapter 5 *Low-overhead UDP Functions*

Read this chapter for a description of the low-overhead UDP functions.

Chapter 6 *The TCP Zero-copy API*

Read this chapter for information on sending and receiving data with the TCP Zero-copy API.

Chapter 7 *ARM-specific Functions*

Read this chapter for details on the sample sources provided as part of the TCP/IP stack. This chapter also details functions, specific to the ARM environment, that these files contain.

Chapter 8 *Miscellaneous Library Functions*

Read this chapter for a description of the assortment of functions to be found in the `\misclib` directory. These functions perform a variety of tasks that are used by the example programs and by the TCP/IP stack.

Chapter 9 *Internal Functions*

Read this chapter for a description of ARP, IP, and ICMP functions.

Appendix A *Error Codes*

Read this appendix to see a list of both the standard `ENP_` error codes you might encounter while porting, and the socket error codes.

Appendix B *Editing ARM Networking .nv Files*

Read this appendix to find out how to edit ARM networking parameters.

Appendix C *Sample Applications*

Read this appendix for information on the sample applications provided, and on how to build and run projects.

Appendix D *The i8255x Ethernet Driver*

Read this appendix for information on the i8255x Ethernet driver, its features and build options, and instructions on porting.

Typographical conventions

The following typographical conventions are used in this book:

`typewriter` Denotes text that may be entered at the keyboard, such as commands, file and program names, and source code.

`typewriter italic`

Denotes arguments to commands and functions where the argument is to be replaced by a specific value.

`typewriter bold`

Denotes language keywords when used outside example code.

italic Introduces special terminology, denotes internal cross-references, and citations.

bold Highlights interface elements, such as menu names. Also used for emphasis in descriptive lists, where appropriate.

Further reading

This section lists publications from both ARM Limited and third parties that provide additional information on porting ARM TCP/IP.

ARM publications

This book contains reference information that is specific to ARM TCP/IP. For additional information, refer to the following ARM publications:

- *Porting PPP Programmer's Guide* (ARM DUI 0143)
- the documentation set for the *ARM Developer Suite (ADS)*
- *ARM Architecture Reference Manual* (ARM DDI 0100).

Other publications

For other reference information, please refer to the following:

- Comer, Douglas E., *Internetworking with TCP/IP: Principles, Protocols, and Architecture*, 3rd Edition, 1995, Prentice-Hall (ISBN 0-13-216987-8).
- Jagger, David, *ARM Architecture Reference Manual*, 1997, Prentice-Hall (ISBN 0-13-736299-4).
- Kernighan, Brian W. and Ritchie, Dennis M., *The C Programming Language*, 2nd Edition, 1988, Prentice-Hall (ISBN 0-13-110370-8).

- RFC 1071, Borman, D., Braden, B. and Partridge, C., *Computing the Internet checksum*, 09/01/1988.
- RFC 1072, Braden, B. and Jacobson, V., *TCP extensions for long-delay paths*, 10/01/1988.
- RFC 1213, McCloghrie, K. and Rose, M., *Management Information Base for Network Management of TCP/IP-based internets: MIB-II*, 03/26/1991.

Feedback

ARM Limited welcomes feedback on both ARM TCP/IP, and its documentation.

Feedback on ARM TCP/IP

If you have any problems with ARM TCP/IP, please contact your supplier. To help us provide a rapid and useful response, please give:

- details of the release you are using
- details of the platform you are running on, such as the hardware platform, operating system type and version
- a small standalone sample of code that reproduces the problem
- a clear explanation of what you expected to happen, and what actually happened
- the commands you used, including any command-line options
- sample output illustrating the problem.

Feedback on this book

If you have any comments on this book, please send email to `errata@arm.com` giving:

- the document title
- the document number
- the page number(s) to which you comments apply
- a concise explanation of your comments.

General suggestions for additions and improvements are also welcome.

Chapter 1
Introduction

The TCP/IP porting sources are provided to enable you to port ARM TCP/IP to other networking environments.

This chapter introduces networking, the ARM porting functions, the requirements for porting ARM TCP/IP, a list of the sample package directories, and an overview of the sample programs provided. This chapter contains the following sections:

1.1 A typical embedded networking stack

Figure 1-1 shows the events that drive a typical embedded networking stack and the responses from the stack:

- the user enters commands
- packets are received from the network
- timers go off.

In each case, a call is made to the stack to handle the event.

In response to these events, the stack:

- makes calls to the system
- sends network packets
- returns data or status information to the calling user
- sets additional timers.

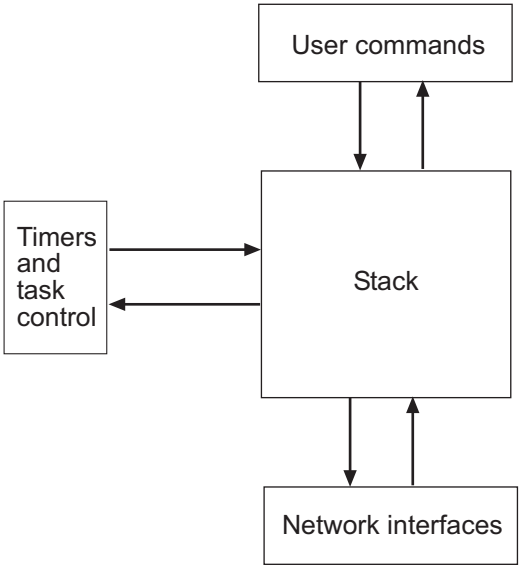

Figure 1-1 Network stack events

In an ideal situation, calls are mapped directly onto the underlying system. For example, the external call from the stack to send a packet has the same syntax as t exported send call from the network interface. However, in a more typical situation, the stack designer does not know what tasking system, user applications, or interfaces are supported in the target system.

The ARM portable stack is designed with simple, generic interfaces. You must create a glue layer that maps this generic interface onto the specific interfaces available on the target system. For example, the stack is designed with a generic `send_packet()` call, and you code a glue function to send the packet using the system network interface. In this example, an ethernet driver, a SLIP driver, and a PPP driver each need different glue functions.

To maximize portability, the stack:

- minimizes the number of calls to glue functions
- uses simple glue functions
- provides detailed documentation
- either uses standard interfaces (such as sockets and the ANSI C library) or provides examples when there is no standard.

The majority of the work in porting a stack is understanding and implementing the glue functions. The ARM TCP/IP stack has the following categories of glue functions:

- *Application Program Interface* (API)
- memory allocation and management
- network hardware interface
- timer and tasking interface.

These functions are discussed in greater detail in Chapter 2 *TCP/IP API Functions*.

1.1 ARM TCP/IP requirements

Before beginning a port, you must ensure that the necessary resources are available in the target environment. There must be:

- a processor with spare CPU capacity with an operating system
- a debug monitor
- RAM
- a network interface.

The exact size of these resources varies depending on the features you plan to implement, the performance you require, and how many simultaneous users you have to support.

The following is a brief summary of services that ARM TCP/IP requires from the system:

- at least one network interface device
- a timer that ticks at least once a second
- memory, processing power, and operating system requirements, which are detailed in this section.

1.1.2 Memory requirements

It is not possible to provide exact memory requirements, however an estimate can be obtained by examining the results provided by the examples. The values provided in Table 1-1 on page 1-5 are from the menus example (see *menus* on page C-5). The code was compiled for the ARM7TDMI processor core in Thumb state with ADS, version 1.0.1 and was optimized for code size rather than speed. All figures exclude C runtime libraries, board support, and the *Menus* application.

The majority of the `ipport.h` build options (such as, routing, `NPDEBUG`, `NET_STATS`, and `DHCP_CLIENT`) are not defined, so the sizes shown in the table below are smaller than if they had been defined. Packet buffers, that are allocated from the heap at initialization time, are not included in these figures.

Note

The values in Table 1-1 on page 1-5 might change with subsequent releases because the code is subject to continuous development.

Table 1-1 ARM7TDMI memory requirements (in bytes) for a basic stack

	Thumb code and read-only data	Read-write data	Zero-init data	Thumb ROM	RAM
IP	6720	8028	408	6804	488
ICMP	664	4	104	668	108
UDP	568	4	16	572	20
TCP	10304	76	172	10380	248
sockets	1652	0	0	1652	0
ARP	1272	20	232	1292	252
Ethernet driver	4303	80	128	4336	208
NV parameters	1336	12	200	1348	212
Totals	26820	276	1260	27096	1536

The compilers place read-only data items, such as strings and constants, into the read-only code area. Therefore, the memory requirements for the code area include these constant data items. When designing your system, you have to estimate the amount of ROM and RAM required:

- The ROM requirement is the sum of the read-only area and the read-write data (the read-write data has pre-initialized values).
- The RAM requirement is the sum of the read-write data and the zero-init data, plus the dynamic memory requirements of the system (for example, network buffers).

TCP and sockets use a large portion of memory. Systems that require only IP and *User Datagram Protocol* (UDP) services (such as routers and SNMP agents) can remove the TCP and sockets layers to reduce memory requirements.

1.2.1 Operating system requirements

The stack requires the following basic services from the target system:

clock tick A clock tick counter must be incremented at regular intervals. See `cticks` in the section on *Timers and multitasking* on page 3-8.

memory access

The standard `malloc()` and `free()` library calls are ideal. If you do not need a TCP or *Dynamic Host Configuration Protocol* (DHCP) server, these functions are called at startup time only and can be replaced with a static or partition-based scheme.

multitasking

The stack must obtain CPU cycles to process received packets (and handle timeouts) on a timely basis. Two methods are discussed in *Task control* on page 3-14.

1.3 Sample package directories

The ARM TCP/IP distribution contains the following directories:

`install_directory\armthumb`

Code common to ARM and Thumb architectures.

`install_directory\docs`

Documentation.

`install_directory\inet`

IP, UDP, and related sources (excluding TCP and sockets), including startup, interface, and buffer management code.

`install_directory\integrator`

Code specific to the Integrator/AP development card fitted with an ARM7TDMI processor and an Intel PRO/100+ Server Adapter.

`install_directory\tcp`

TCP and sockets source files.

`install_directory\chargen`

Demonstration of a simple server.

`install_directory\loopback`

MAC and PPP loopback benchmark code.

`install_directory\maildemo`

A simple client that uses SMTP to send an email message.

`install_directory\menus`

Menuing system, useful for debugging.

`install_directory\emailer`

Source code for the Email Alerter embedded SMTP client. This package allows an embedded system to send email messages.

`install_directory\misclib`

Miscellaneous support routines required by the stack.

`install_directory\uHAL`

μHAL and PCI libraries and header files from the ARM Firmware Suite. These libraries provide the low-level board support for the Integrator/AP platform.

1.2 Sample programs

The sample code compiles with ADS 1.0.1. Unless you are familiar with TCP/IP and are comfortable working with complex networking code, it is recommended that you compile the sample programs and experiment with them before you port your application. Instructions for compiling sample programs are in Appendix C *Sample Applications*.

Chapter 3
TCP/IP Porting

This chapter discusses porting the TCP/IP stack to a new environment. It is assumed that the stack is being ported to a small, embedded system with a network interface and that ADS 1.0 is available. This chapter contains the following sections:

3.1 Porting procedure

To create a working version of the TCP/IP stack on your target system:

1. Copy the portable source files into your development environment (see *Portable and nonportable files* on page 3-3).
2. Create a version of `ipport.h` (see *Creating the IP port file* on page 3-4) and compile the portable sources.
3. Write and compile the necessary code for the glue layers (see *Coding the glue layer* on page 3-14).
4. Specify the IP address information (see *Specifying IP addresses* on page 3-18).
5. Add your own code to use the stack.
6. Build a target system image, test, and debug (see *Testing the TCP/IP port* on page 3-20).

3.2 Portable and nonportable files

There are two types of files provided in the distribution:

- portable files that can be compiled and used on any target system without modification
- nonportable or port-dependent files that must be modified or replaced for different target systems.

3.2.1 Portable files

Portable files are the IP and TCP stack source files that must not need to be modified in the course of a normal port. If you need to modify one of these files, discuss it with ARM technical support staff first, as we may be able to suggest an alternative.

These files are in the `\inet` and `\tcp` directories under your installation directory.

3.2.2 Nonportable files

All other files in the sample package contain nonportable glue layer code that must be modified, replaced, or omitted.

The support functions required for basic operation of the stack are covered in Chapter 2 *TCP/IP API Functions*.

You must also provide at least one network interface that conforms to the specification described in *Network interfaces* on page 2-14. The sample sources come with a sample Ethernet network interface driver.

The ARM TCP/IP stack and related applications, such as DHCP and SNMP, generally have one C file and one include file containing all the nonportable code for that module. These files have the generic name `moduleport.x`, where *module* is the module name (for example, `tcp` or `http`), and `x` is either `.c` or `.h`. For example, the TCP directory has the files `tcpport.c` and `tcpport.h`.

The `ipport.h` file is kept with your application code, allowing different applications to use different TCP/IP stack configurations. All other source files in the TCP directory are intended to be fully portable.

3.3 Creating the IP port file

This section contains the following information:

3.3.1 The ipport.h file

The `ipport.h` file contains the port-dependent definitions for the IP layer, and the architectural definitions for all the IP-related code. It also controls CPU architectures (big-endian or little-endian), compiler options, and optional features (DHCP, multiple interfaces, and IP routing support).

Caution

A mistake in this file (such as using big-endian in place of little-endian) can create severe problems, so it is important to set this file up correctly.

You must create a version of the IP port file before you compile the portable TCP/IP stack files, and must `#include` the `ipport.h` file in every C file of every module throughout the TCP/IP software.

3.3.2 Standard macros and definitions

The ARM TCP/IP stack expects `TRUE`, `FALSE`, and `NULL` to be defined in `ipport.h`. Typically, the best way to do this is to include the standard C library file `stdio.h` in `ipport.h`. If `stdio.h` is impractical to use or not available on your system, the following example works in most environments:

```
#ifndef TRUE
#define TRUE -1
#define FALSE 0
#endif
#ifndef NULL
#define NULL (void*)0
#endif
```


3.3.3 CPU architecture

Four common macros from Berkeley UNIX are used for doing byte order conversions between different CPU architecture types:

- `htons()`
- `htonl()`
- `ntohs()`
- `ntohl()`.

You can use these functions as either macros or functions. They accept 16-bit and 32-bit quantities as shown and convert them from network format (big-endian) to the format supported by the local system.

If your system is using the ARM processor in big-endian mode, these macros can return the variable passed, for example:

```
#define htonl(l) (l)
#define htons(s) (s)
#define ntohl(l) (l)
#define ntohs(s) (s)
```

If your system is using the ARM processor in little-endian mode, the byte order must be swapped. For `htonl()` and `ntohl()`, use the `lswap()` function provided in the `\armthumb` directory (see *Sample package directories* on page 1-7). For `htons()` and `ntohs()`, use a byte-swapping macro, as shown below:

```
#define htonl(l)    lswap(l)
#define htons(s)    ((u_short)(((u_short)(s) >> 8) |
                               ((u_short)(s) << 8)))
#define ntohl(l)    lswap(l)
#define ntohs(s)    htons(s)
```

3.3.4 Pre-emption and protection

You must define primitives in order to protect sections of code that must not be interrupted or pre-empted (see *Implementing pre-emption and protection* on page 3-16).

The critical section protection scheme is typically used on embedded systems that lack a multitasking capability:

```
void ENTER_CRIT_SECTION();  /* enter critical section */
void EXIT_CRIT_SECTION();   /* exit critical section */
```

Those systems are described in detail in *ENTER_CRIT_SECTION() and EXIT_CRIT_SECTION()* on page 2-6.

The lock net resource macros are typically used on real-time kernels, such as VRTX and VxWorks:

```
void LOCK_NET_RESOURCE();   /* start re-entrance protection */
void UNLOCK_NET_RESOURCE(); /* end re-entrance protection */
```

3.3.5 Debugging aids

You must include the following macros in your functions to provide support while you are debugging your code:

- *dtrap*
- *initmsg() and dprintf()* on page 3-7
- *NPDEBUG* on page 3-7.

dtrap

This macro is called by the stack code when it detects a situation that should not be occurring. The intention is for the `dtrap()` macro to invoke whatever debugger may be in use by the programmer. In this way, it acts like an embedded breakpoint. The stack code can continue executing after a `dtrap()`, but the `dtrap()` typically indicates that something is wrong with the port.

Note

Products based on this code should not be shipped until all calls to `dtrap()` have been removed from the code. You can redefine `dtrap()` to a null macro to slightly reduce code size.

initmsg() and dprintf()

The `initmsg()` and `dprintf()` macros have the same function and syntax as `printf()`. They have different names so their output can be redirected to different locations, or so they can be individually disabled.

The `initmsg()` macro is called by various stack functions to print function status messages during initialization. These messages are for information and are not warnings.

The `dprintf()` macro is used throughout the stack code to print warning messages when something seems to be wrong.

In most ports, these can both be mapped to `printf()` while the product is under development.

Note

Mapping `initmsg()` and `dprintf()` to `printf()` works with the ADS, but performance of the stack is severely affected by the time taken to transfer debugging information across the RDI link to the ARM debugger.

A more efficient alternative is to use low-level functions to replace `putchar()` and `printf()` and to write output to a UART or other console device. The file `\misclib\ttyio.c` contains an example `dprintf()` function that can be used in conjunction with `\integrator\uartio.c` for this purpose.

```
#define initmsg printf  /* Same parms as printf. Called at */
                        /* boot time */
#define dprintf printf  /* Same parms as printf. Called */
                        /* during run time */
```

For some products, it may be desirable to define these to a null macro before releasing.

```
#define initmsg         /* define to nothing */
#define dprintf         /* define to nothing */
```

NPDEBUG

Defining the `NPDEBUG` macro causes the debug code to be compiled into the application. The debug code performs tasks, such as checking for valid parameters and sensible configurations during runtime. The debug code invokes `dtrap()` or `dprintf()` to inform the programmer of detected problems. To make use of this feature, make sure `NPDEBUG` is defined during development.

```
#define NPDEBUG 1   /* enable debug checks */
```


3.3.6 Timers and multitasking

The ARM TCP/IP stack requires a clock tick for TCP and ARP timers. The stack depends on an unsigned long variable named `cticks`.

The `cticks` variable should be regularly incremented (between 5 and 100 times per second) by the port code, wrapping back to 0 after reaching `0xFFFFFFFF`. The stack code uses the `TPS` macro to adjust `cticks` (ticks per second) for actual time. You must define this in `ipport.h`, for example:

```
#define TPS     50  /* cticks per second*/
```

An example using µHAL timers is available in `\integrator\clock.c`. The example sets the timer to match the value assigned to `TPS` in `ipport.h`.

If you are testing the stack using the ARMulator, you can define `cticks` to call the `clock()` library function instead:

```
#define cticks (clock())
extern long clock(void);
```

If you use `clock()` to provide the value for `cticks`, you must define `TPS` to be `100` in the `ipport.h` file.

3.3.7 Stack features and options

The stack assumes that you have at least one device for sending and receiving network packets. These devices are usually hardware devices, such as Ethernet or serial ports, but they can be logical devices, such as loopback drivers or inter-process communication software.

Most IP stacks on embedded systems support only one device. However, devices like routers need two or more. The ARM stack supports multiple logical devices and has been used with up to three. The structures to manage these devices are statically allocated at compile time, so the maximum number of devices the system uses at run time must be set in `ipport.h`:

```
/* define the maximum number of hardware interfaces */
#define MAXNETS      2  /* maximum entries of nets[] array */
```


3.3.8 Optional compilation switches

The lists of optional compilation switches in this section can be defined in `ipport.h`. The options must be commented out if they are not required. Some of these switches add hooks. The switches available depend on the products installed.

Note

Setting the defined value to 0 does not disable the feature. You must comment out the definition to disable it.

General switches

`INCLUDE_ARP`
If defined, the stack uses the ARP protocol to perform physical address resolution on those interfaces that support ARP.

If not defined, the ARP protocol is not performed. ARP is required for Ethernet operation.

`FULL_ICMP`
If defined, the stack implements the entire ICMP protocol.

If not defined, the stack implements only the ping (ICMP ECHO) protocol.

`INCLUDE_TCP`
If defined, the stack includes support for the TCP protocol.

`TCP_ZEROCOPY`
If defined, the stack includes support for the TCP Zero-Copy API extensions.

`NPDEBUG`
If defined, debug messages are displayed on the system console.

`IP_FRAGMENTS`
If defined, the stack attempts to reassemble IP packet fragments that are destined for the target system.

If not defined, the stack silently discards fragmented IP packets that are destined for the target system.

This option also controls whether or not the stack generates fragmented IP packets.

`NB_CONNECT`
If defined, the stack includes code to support non-blocking connection attempts.

`IP_ROUTING`
Controls whether or not IP layer packet routing is performed on multihomed hosts.

MULTI_HOMED	This constant must be defined if the target system can have more than one physical interface, for example, an Ethernet and a serial link. This must be undefined on targets with a single interface to minimize target memory usage.
NET_STATS	If defined, the stack includes the code and data structures that allow stack performance and other statistics to be displayed.
IP_LOOPBACK	If defined, the stack can loopback packets destined for the IP loopback address at the bottom of the IP layer, without being queued to a network interface.
MAC_LOOPBACK	If defined, the stack can loopback packets destined for the IP loopback address via a separate loopback network interface.
NO_UDP_CKSUM	If defined, UDP checksums are not generated for transmitted UDP packets and are not verified on received UDP packets.
MEM_LIBS	The stack includes implementations of several C library functions like `memcpy()`, `memset()`, and `memcmp()`, for targets whose C libraries do not include these functions. Define this constant if your C library does not include implementations of these functions.
INICHE_LIBS	The stack includes implementations of several C library functions that operate on strings (for example, `strlen()`, `strcpy()`) for targets whose C libraries do not include these functions. Define this constant if your C library does not include implementations of these functions.
NATIVE_PRINTF	The `misclib` directory contains an implementation of formatted output functions that perform a function similar to `printf()`. If you intend to use these functions instead of the formatted output function provided by your C compiler, this constant should be undefined. If you intend to use the `printf()` from your C compiler for formatted output, you must define this constant.
INCLUDE_NVPARMS	If defined, the stack includes code that can parse a configuration file to set the various parameters for the stack, such as IP addresses and DNS configuration.
MONITOR_ALLOCS	This causes `npalloc()` and `npfree()` to track memory allocation, check for freeing memory that has not been allocated, check for freeing memory that has already been freed, and check for memory heap corruption.
USE_I8255X	If defined, the stack uses the Intel 82559 Ethernet driver.

VFS_FILES	If defined, the stack uses the Virtual File System routines found in the vfs directory.
HT_LOCALFS	If defined, the stack accesses files on the local system using fopen / fread. This can be used with VFS_FILES.
WANT_PACKET	If defined, this includes a routine that can be called by the Ethernet *Interrupt Service Routine* (ISR) to determine if a packet is potentially of interest to the stack. This allows the ISR to discard uninteresting packets instead of queuing them, which helps reduce the number of packet resources being consumed, but at the cost of some code space and interrupt service time.
IN_MENUS	If defined, the stack includes the diagnostic menu system.
MENU_HISTORY	If defined, the diagnostic menu system supports a history mechanism. You must #define MENU_HISTORY to be the number of commands to be remembered. For example, the following line allocates ten slots in the history array: `#define MENU_HISTORY 10`
UDPSTEST	If defined, the Sockets based UDP echo client and server are included in the target.
DNS_CLIENT	If defined, the stack includes code to perform DNS lookups of host names.
PING_APP	Define this constant if the target system generates ICMP ping/echo requests. This constant is not required for the client to respond to echo requests from other hosts.
DHCP_CLIENT	If defined, the stack implements the client side of the DHCP protocol during system startup.
SMTP_ALERTS	If defined, the stack includes code to send email alerts to a configured user upon the detection of various runtime exception and error conditions.
TCP_ECHOTEST	If defined, the TCP echo client and server are included in the target.

ARM PPP

The following are available if you have ARM PPP:

`USE_PPP` If defined, the stack implements the PPP protocol.

`USE_MODEM` If defined, the stack supports call setup and hangup using a Hayes-compatible modem.

`RAS_DIRECT` If defined, the stack supports direct cable connection networking to Microsoft RAS servers. This option requires `USE_PPP` and `USE_MODEM` to be defined as well.

ARM Embedded Web Server

The following switch is only functional if you have the ARM Embedded Web Server software:

`WEBPORT` If defined, the stack includes the ARM HTTP server.

ARM SNMP Agent

The following switches are only functional if you have the ARM SNMP Agent software:

`INCLUDE_SNMP` If defined, the stack includes the ARM SNMP agent.

`SNMP_SOCKETS` If defined, the stack implements the SNMP agent using the standard Sockets API. This is useful if you wish to use the ARM SNMP product without the ARM TCP/IP product.

ARM FTP Server

The following switches are only functional if you have the ARM FTP Server software:

`FTP_SERVER` If defined, the stack includes an FTP server.

`FTP_CLIENT` If defined, the stack includes an API that allows the calling application to generate the client side of the FTP protocol.

ARM DHCP Server

The following switch is only functional if you have purchased the ARM DHCP Server software:

`DHCP_SERVER` If defined, the stack includes the ARM DHCP server.

ARM Telnet Server

The following switch is only functional if you have purchased the ARM Telnet Server software:

`TELNET_SVR` If defined, the stack includes the ARM Telnet Server.

ARM RIP

The following switch is only functional if you have purchased the ARM Telnet Server software:

`RIP_SUPPORT` If defined, the stack includes the ARM RIP functionality.

ARM TFTP Server

The following switches are only functional if you have the ARM TFTP Server software:

`TFTP_SERVER` If defined, the stack includes the ARM TFTP Server.

`TFTP_CLIENT` If defined, the stack includes an API that allows the calling application to generate the client side of the TFTP protocol.

3.4 Coding the glue layer

After you have edited your `ipport.h` file (*Creating the IP port file* on page 3-4), you must code the glue layers in `ipport.c`. These functions map the generic service requests made by the stack to the specific services provided by your target system.

Many requests are handled through mappings in `ipport.h`. The remainder must be implemented as a minimal layer of C code and as interface functions to drive any hardware devices required. In the example package, some of these are collected in `ipport.c`, and the rest are in separate source files within the `\armthumb` and `\integrator` directories.

Typically, the most complex part of the glue layers is the network hardware interface, described in *Network interfaces* on page 2-14. The `ipport.c` file exports a function called `prep_ifaces()`, that:

- initializes the pre-allocated network structures to point to the interface functions
- fills in hardware specific parameters
- sets up MIB structures.

You might have to modify this function to reflect your own interfaces.

Most of the functions you require in order to code the glue layer are described in Chapter 2 *TCP/IP API Functions*. Every function described there must be either coded or mapped directly onto a system function using a macro in `ipport.h`.

3.4.1 Task control

The TCP/IP stack must obtain CPU cycles to process received packets and handle timeouts on a timely basis. This is achieved using a superloop that regularly polls using a central function.

Like some simpler embedded systems, there is no real multitasking system available to the example code. The sample programs obtain control after they are loaded and run until the user tells them to stop. Internally they are in an infinite loop waiting for new input to act on. This internal loop is referred to as the *superloop*.

The example works as follows:

1. At the end of the `main()` function, the example calls all the initialization functions and enters an infinite loop calling the nonportable function `tk_yield()`.
2. The `tk_yield()` function polls all the linked modules (for example, the IP stack, servers, and modem drivers) and returns.
3. The linked modules are called through a portable function that checks for work to be done by that module, processes any work, and returns.

In the case of the IP stack, `tk_yield()` calls the nonportable `packet_check()` to guard against re-entry, which calls `pkt_demux()`, the portable function that dequeues received packets.

This superloop technique is well suited to taskless systems. The drawback is that CPU cycles are wasted by polling functions that have no work to do.

3.4.2 TCP

The TCP portion of the stack implements two functions that allow it to wait if resources, such as free buffers, are unavailable or the remote host responds too slowly. These functions are:

- `tcp_sleep()`
- `tcp_wakeup()`.

You must map these functions onto the block or sleep function your OS provides.

If you are not using TCP in your port, you do not have to implement these functions.

The `tcp_sleep()` function takes as its argument a pointer (usually to a socket data buffer) on which to block. This means the calling TCP function is waiting for data. At the very least, the system should give the net task a chance to process more incoming packets before returning from the call to `tcp_sleep()`.

Whenever data arrives for such a buffer, the TCP code calls `tcp_wakeup()`.This means that for optimum efficiency, a call to `tcp_sleep()` can block until the related call to `tcp_wakeup()` is received.

The stack calls the `tcp_wakeup()` function with the same pointer that `tcp_sleep()` was called with. When `tcp_wakeup()` is called with a particular pointer value, your code must ensure that all of the tasks that called `tcp_sleep()` with that value are made runnable.

If `tcp_sleep()` returns before the `tcp_wakeup()` call is made, the calling TCP code loops back to retest the buffer condition and calls `tcp_sleep()` again.

In the example port, `tcp_sleep()` calls the superloop function, `tk_yield()`, and `tcp_wakeup()` is a null operation. This is the simplest possible round-robin process. For details on how this is done, refer to the source code in `tcpport.c`. Most RTOS products support a type of call that lets the round-robin scheduler spin all the tasks once and then return. This is typically what `tcp_sleep()` does.

If you are familiar with the UNIX kernel `sleep()` and `wakeup()` functions, you can see that `tcp_sleep()` and `tcp_wakeup()` can be mapped directly to the UNIX `sleep()` and `wakeup()` functions.

3.4.3 Implementing pre-emption and protection

You must implement a process control method to protect the internal structures of the stack from being corrupted by reentrant code. The following is an example problem scenario:

1. The IP stack code is in the process of adding an item to a queue as the result of a socket call by a user application and has made local copies of some of the structure internals of the queue.
2. An IP packet arrives and the net task wakes up, pre-empting the user task and modifying the same queue copied in step 1.
3. The user task resumes and finishes adding the item to the queue, corrupting the structure of the queue when it writes back its local copies of the internal structures of the queue.

The ARM TCP/IP stack provides two separate methods of protecting itself against this kind of problem:

- *The critical section method*
- *The network resource lock method* on page 3-17.

You must select the method that best fits your target system and ensure the appropriate macros are implemented in `ipport.h`.

The critical section method

A critical section refers to a sequence of code that must be allowed to complete without interruption. This code is bracketed between macros or C calls designed to ensure that the code is not interrupted. For the ARM stack these functions are:

- `ENTER_CRITICAL_SECTION()`
- `EXIT_CRITICAL_SECTION()`.

Critical sections are typically the preferred method of protection on general purpose systems like DOS or UNIX. On UNIX kernels, these macros can be mapped directly to the `splnet()` and `splx()` calls. Typically, ARM ports implement these macros by turning off interrupts. Refer to the example source in `\integrator\irq.c`.

`ENTER_CRITICAL_SECTION()` and `EXIT_CRITICAL_SECTION()` must handle nesting correctly. As an example, `ENTER_CRITICAL_SECTION()` could disable interrupts and increment a count of how many times it has been called. `EXIT_CRITICAL_SECTION()` decrements that count and then only re-enables interrupts when the count has been decremented to zero.

See *ENTER_CRIT_SECTION() and EXIT_CRIT_SECTION()* on page 2-6 for more information.

The network resource lock method

The resource lock method of protection is provided as an alternative to critical section protection. This method was created for systems with strict real-time requirements that want to run the networking protocols at a low priority, and want to pre-empt them at any time. Resource locking assigns three system resource IDs to the IP stack. These work as mutex (mutually exclusive) semaphores. The macros for this are:

- `LOCK_NET_RESOURCE()`
- `UNLOCK_NET_RESOURCE()`.

See *LOCK_NET_RESOURCE() and UNLOCK_NET_RESOURCE()* on page 2-7 for more details.

3.5 Specifying IP addresses

Before you can test your stack, you must set up some basic IP addressing information. The ARM stack offers the latest protocols for setting up IP addresses, but it still requires additional information from both the porting programmer and the end user.

3.5.1 Porting programmer IP issues

You must first assign a valid IP address to each interface.

Caution

You must assign these addresses as part of the port. If the IP address values are not correctly set, your stack might not work and you might even disable other users on the net. If you do not know which values to use, ask your network administrator or refer to *Internetworking with TCP/IP*, by Douglas E. Comer, on IP addressing.

Fill in the `n_ipaddr` member of the net structure for each interface. (The net structure is discussed in detail in *Network interfaces* on page 2-14.) This must be done before or during the network interfaces `n_init()` call. You must also set the fields for `n_defgw` (default router) and `snmask` (subnet mask). These three values are collectively referred to as the IP addressing information.

Note

In little-endian systems, these addresses are stored in network byte order (big-endian format) rather than little-endian. Refer to the `n_ipaddr` setup code in `\inet\macloop.c` for a portable example that works on both big-endian and little-endian architectures.

3.5.2 End user IP issues

A product issue you will encounter is how the end user assigns the IP addresses. Traditionally, the end user was required to sit down at a console attached to the device, usually a serial terminal attached to an RS-232 port, and key in an IP address, subnet mask, default router, and possibly other information. The IP address information was stored in permanent storage (NVRAM or disk) and read each time the machine was booted.

Although it is still a good idea to support manual IP address assignment, more recent IP technology has developed several easier ways to do this, most notably DHCP. For a description of the ARM DHCP client functions, see Chapter 3 *DHCP Client Functions*.

3.6 Testing the TCP/IP port

When your `ipport.h` file is set up and your glue layers are coded, you are ready to test your stack. The traditional first test of most IP stacks is `ping`, the popular term for ICMP echo packets. You must connect your hardware to the network and `ping` from a remote machine. At a DOS or UNIX shell prompt, enter:

```
ping number
```

where *number* is an IP address. Use the number you assigned to the network interface of your stack (in *Specifying IP addresses* on page 3-18).

When you use `ping`, it sends a network packet to the ARM stack, which echoes it back. This indicates that:

- the packet was transferred from the net media to your interface
- the IP information for your interface is properly configured
- your IP layer is receiving from the interface
- the ICMP layer has attached to IP
- ICMP can send to IP
- IP can send to the interface
- the interface can send on the physical net.

If you are using Ethernet, ARP has also worked.

If `ping` works, most of your port is complete.

The other tests you should run depend on your product. ARM also sells FTP servers, SNMP agents, and Web Servers for embedded systems. If you are using any of these packages, refer to the documentation provided for implementation and testing information.

Once you have been able to use ping to verify the basic operation of the stack, you might want to use the `menus` sample program to more fully explore the stack, and to test the TCP and UDP modules. The sample programs are described in detail in Appendix C *Sample Applications*.

Chapter 2
TCP/IP API Functions

This chapter describes the functions the user must provide to port the ARM TCP/IP stack. Refer to the code provided with the software for examples. It contains the following sections:

2.1 User-provided TCP and IP functions

This section describes the TCP and IP functions that you must implement as part of porting the ARM TCP/IP stack. A description of how to use the function is given in each case.

In the sample package, these functions are either mapped directly to system calls by way of macros in `ipport.h`, or they are implemented in `ipport.c`, `tcpport.c`, or in files in the `\armthumb`, `\integrator` and `\misclib` directories. Many of these implementations map directly onto the system to which you are porting. Others need extensive modification or complete rewrites.

Refer to Chapter 3 *TCP/IP Porting* for the complete TCP/IP porting procedure.

The functions are as follows:

1.0.1 cksum()

This function returns a 16-bit Internet checksum of the buffer.

The algorithm for this is described in RFC 1071.

Syntax

```
unsigned short cksum(unsigned short *buffer, unsigned word_count)
```

where:

`buffer` Is the pointer to the buffer to checksum.

`word_count` Is the number of 16-bit words in the buffer.

Return value

The `cksum` function returns the 16-bit checksum.

Usage

Both C language and ARM assembly language versions of this function are provided with the sample package. The C version is in `\inet\ccksum.c` and the assembler version is in `\armthumb\cksum.s`. You must include only one of these files in your project.

The C version is included to provide an insight into what is going on in the assembler versions. A significant amount of TCP/IP stack processor time is spent in the `cksum()` function, so the optimized assembler versions must be used.

1.0.2 dprintf() and initmsg()

Both `dprintf()` and `initmsg()` are functionally the same as `printf()`. Both are called by the stack code to inform the programmer or end user of system status. The `initmsg()` function prints normal status messages at initialization time and `dprintf()` prints error and warning messages during runtime.

Syntax

```
void dprintf(char *fmt, …)
```

```
void initmsg(char *fmt, …)
```

where:

`fmt` Is a format string like `printf()`.

… Is an argument list, as described by `fmt`.

Return value

None.

Usage

You can either define these functions to use `printf()` in `ipport.h` or you can write your own implementation. See the sample code in `\misclib\ttyio.c` for a sample implementation.

The `ttyio.c` implementation of `dprintf()` uses of the function `dputchar()` to perform its output. By default, `dputchar()` is implemented as part of the UART driver, to allow debug output to be sent to a serial terminal. If you have other debug channels, for example a video card, then debug may be redirected by recoding `dputchar()`.

See also the detailed description in *Debugging aids* on page 3-6.

1.0.3 dtrap()

This function can enter a debugger when it is called.

Syntax

```
void dtrap(void)
```

Return value

None.

Usage

See the detailed description in *Debugging aids* on page 3-6.

1.0.4 ENTER_CRIT_SECTION() and EXIT_CRIT_SECTION()

These two functions protect a sequence of code that must be allowed to complete without interruption (see *Implementing pre-emption and protection* on page 3-16).

Syntax

```
void ENTER_CRIT_SECTION(void *ptr)
```

```
void EXIT_CRIT_SECTION(void *ptr)
```

where:

`ptr` Refers to a memory location specific to this critical section. The `ptr` parameter might be used by your function to identify which critical section is being entered and released, and to confirm that nested critical sections are exited in the correct order.

Return value

None.

Usage

Typically these functions disable and re-enable interrupts. On UNIX-like systems, they can be mapped to the `spl()` primitive. Examples for the Integrator/AP are provided in the sample code.

Refer to *The critical section method* on page 3-16 for more information on critical sections.

1.0.5 LOCK_NET_RESOURCE() and UNLOCK_NET_RESOURCE()

These two functions are used by the system to preserve mutual exclusion on important data structures in much the same way as the ENTER_CRIT_SECTION() and EXIT_CRIT_SECTION() functions described above. Resource locking is required by some RTOS implementations in order to guarantee minimum latency for time critical tasks. If you are porting ARM TCP/IP to such an environment, you must map LOCK_NET_RESOURCE() and UNLOCK_NET_RESOURCE() onto the appropriate calls for your RTOS.

Syntax

```
void LOCK_NET_RESOURCE(int resourceid)

void UNLOCK_NET_RESOURCE(int resourceid)
```

where:

resourceid — Is the system resource identifier to be locked. Three such identifiers are required by the TCP/IP stack:

- NET_RESID — Protects the majority of the data structures on the TCP/IP stack.
- RXQ_RESID — Protects the received packet queue data structure.
- FREEQ_RESID — Protects the free packet queue data structure.

These must be #defined in ipport.h and mapped onto suitable resource identifiers for your operating system.

Return value

None.

Usage

The LOCK_NET_RESOURCE() function must block until the resource identified by resourceid is available. When the resource lock is available, LOCK_NET_RESOURCE() must lock it and return. While waiting for the lock to become available, the task scheduler must be allowed to run other tasks.

Testing and setting the lock must be an atomic operation to prevent two tasks from believing that they have both locked the same resource. If you are using resource locking implemented within an RTOS, this has already been done for you. Otherwise, this is usually implemented by turning off all interrupts while the test and set operations are performed.

The UNLOCK_NET_RESOURCE() function must unlock the resource identified by resourceid and therefore allow any tasks waiting for this resource to continue execution. Refer to *Pre-emption and protection* on page 3-6 for more information.

2.1.1 npalloc()

This function allocates the dynamic memory for the IP stack. The syntax for this function is exactly the same as the standard C library call, malloc(), except that memory returned by npalloc() is assumed to be pre-initialized to all zeros. In this respect npalloc() is like calloc().

Syntax

```
void *npalloc(unsigned size)
```

where:

size Is the size in bytes of the memory to be allocated.

Return value

Returns a pointer to the block allocated, or NULL if no memory is available.

Usage

If your embedded system already supports standard calloc() calls, add the following line to ipport.h:

```
#define npalloc(size)      calloc(1,size)
```

If your system does not support calloc(), you must implement it. A description of how this function works and sample code is available in *The C Programming Language*.

The great majority of the calls to npalloc() are made at initialization time. Only the UDP and TCP layers require these calls during runtime. Some ports with severe memory shortages have modified these layers to use pre-allocated blocks of static memory rather than implementing fully functional npalloc() and npfree() functions, but this invariably involves more work and puts limits on the number of simultaneous connections that can be supported.

You may also want to add debugging facilities in order to help detect any memory leaks within the system. A sample implementation of a simple debugging npalloc() is in \misclib\memman.c.

2.1.2 npfree()

This function frees dynamic memory for the IP stack. The syntax for this function is exactly the same as the standard C library call, `free()`.

Syntax

```
void npfree(void *ptr)
```

where:

`ptr` Points to a block of memory that was previously allocated by `npalloc()`.

Return value

None.

Usage

If your embedded system already supports standard `free()` calls, add the following line to `ipport.h`:

```
#define npfree(ptr)        free(ptr)
```

If your system does not support `free()`, you must implement it. See *The C Programming Language* for a description.

You might want to add some debugging facilities in order to help detect memory leaks within the system. A sample implementation of a simple debugging `npfree()` is in `\misclib\memman.c`.

1.2.6 panic()

This function is called if the stack detects a fatal system error.

Syntax

```
void panic(char *msg)
```

where:

`msg` Is a short message describing the fault.

Return value

Generally there is no return from this function. However, it is sometimes useful to allow a return under the control of a debugger.

Usage

The task performed by this function varies with the implementation. In a testing or development environment, it can print messages, start debuggers, or perform other debugging tasks. The system designer must decide what action is taken in an embedded implementation. One possibility is to restart (warm boot) the system. It is recommended that you do not continue execution after `panic()` has been called.

1.2.7 prep_ifaces()

This call prepares the `nets[ ]` structures for the network interfaces used by this port.

Syntax

```
int prep_ifaces(int ifIndex)
```

where:

`ifIndex` is the index to be used for the first `nets[ ]` structure.

Return value

Returns an index to the `nets[ ]` structure for the next interface to be set up. If no interfaces were set up, the returned value is the same as the passed value.

Usage

The integer passed is the `nets[ ]` index of the first interface to set up. This is always 0 in standard ports, but `nets[0]` has a dedicated use in some customized versions. If you are setting up multiple interfaces, you must set `nets[X]` for each interface. If the interfaces are similar, using a **`for`** loop to set them may be appropriate.

The appropriate `nets[ ]` members must be filled in. See *Network interfaces* on page 2-14 for full details. A sample implementation of `prep_ifaces()` can be found in `\inet\ipport.c`.

2.1.3 tcp_sleep()

This function is called from the TCP code when a TCP operation is blocked by a temporary lack of resources, typically a lack of free buffers.

Syntax

```
void tcp_sleep(void *ptr)
```

where:

`ptr` Is the memory address of some structure relevant to the current task. The `tcp_wakeup()` function is called from elsewhere in the stack to restart this task. The `tcp_wakeup()` function is called with the same memory address.

Return value

None.

Usage

This function must allow the tasking system to attempt to run each task at least once and then return. The calling code then retests the condition and:

- proceeds
- times out
- calls `tcp_sleep()` again.

See the detailed description of this in *TCP* on page 3-15.

1.3.8 tcp_wakeup()

This function is called by the TCP/IP code when an incoming packet or timeout causes a previously blocked process to become runable.

Syntax

```
void tcp_wakeup(void *ptr)
```

where:

`ptr` Is the memory address that the sleeping process passed to `tcp_sleep()`. All tasks that called `tcp_sleep()` with this address is woken.

Return value

None.

Usage

This function must cause all processes that were sleeping because of a call to `tcp_sleep()` with the same value for `ptr` to be marked as runable. In a superloop system, `tcp_wakeup()` performs no action because `tcp_sleep()` does not block. Instead, `tcp_sleep()` calls the nonportable superloop function, `tk_yield()`. `tcp_wakeup()` can be called even if no process is currently blocked in `tcp_sleep()`.

2.2 Network interfaces

Network interfaces are described to the ARM stack by the NET structure in the file \inet\net.h. One of these structures is statically allocated for each MAC link the stack uses (see the definition of MAXNETS in ipport.h). You must write the functions listed below for the hardware used in your target system.

The network interface functions are:

n_close()	Net close function.
n_init()	Net initialization function.
n_reg_type()	Register a MAC type, for example, 0x0800 for IP.
n_stats()	Print net statistics.
raw_send()	Send data on media.
pkt_send()	Send data on media.

You must also provide a packet receive mechanism that takes received packets and places them in the rcvdq queue. The receive function obtains its data buffers as PACKET structures obtained by calls to pk_alloc().

The remainder of this section describes both the NET structure and the functions in detail. The functions are as follows:

- *n_close()* on page 2-17
- *n_init()* on page 2-18
- *n_reg_type()* on page 2-20
- *n_stats()* on page 2-21
- *pkt_send()* on page 2-22
- *raw_send()* on page 2-25.

2.0.9 The NET structure

The NET structure is defined in \inet\net.h. An array of NET structures describes the network interfaces that the IP stack is to use. Each NET structure describes one such interface, as in Example 2-1.

Example 2-1

```
/* The NET struct has all the actual interface characteristics that are visible */
/* at the internet level and has pointers to the interface handling functions. */

struct net {
int (*n_init)(int);            /* net initialization function */
                               /* MAC drivers can set one of the next two for */
                               /* sending; other should be left NULL */
int   (*raw_send)(struct net *, char *, unsigned);  /* put raw data on media */
int   (*pkt_send)(struct netbuf *);                 /* send packet on media */
int   (*n_close)(int);                              /* net close function */
int   (*n_reg_type)(unshort, struct net *);         /* register a MAC type, */
                                                    /* 0x0800 for IP */
void  (*n_stats)(int iface);   /* per device-type (ODI, pktdrv) statistics dump */
int   n_lnh;                   /* net's local net header size*/
int   n_mtu;                   /* net's largest legal buffer size */
ip_addr  n_ipaddr;             /* interface's internet address */
int      n_snbits ;            /* number of subnet bits */
ip_addr  snmask;               /* interface's subnet mask */
ip_addr  n_defgw;              /* the default gateway for this net */
ip_addr  n_netbr;              /* our network broadcast address */
ip_addr  n_netbr42;            /* our (4.2BSD) network broadcast */
ip_addr  n_subnetbr;           /* our subnetwork broadcast address */
unsigned n_hal;                /* Hardware address length */
unsigned n_type;               /* interface type ETHERNET, PPP, SLIP,... */
char *   n_haddr;              /* pointer to hardware address, size=n_hal */
IFMIB    n_mib;                /* pointer to interface(if) mib structure */
void *   n_local;              /* pointer to custom info, null if unused */
};

typedef struct net *NET;
```

The NET structure contains:

- Six pointers to functions within the interface device driver. The functions are described in detail throughout this section.

- Information about the interface device driver. The four media information values (described below) must be initialized either when your network interface is prepared or when its `n_init()` function is called.

  `n_lnh` Describes the length (in bytes) of the local network header. This is the amount of space that is reserved by the IP layer at the front of a packet buffer so the device driver layer can insert any addressing information required by the networking hardware. This is 16 for Ethernet.

  `n_mtu` Contains the size of the largest packet that can be transmitted by the network hardware. For Ethernet, this is 1500 + `n_lnh`.

  `n_hal` Contains the length of a hardware address (MAC address) for the network hardware. This is 6 for Ethernet.

  `n_haddr` This points to the hardware address for this interface.

- IP addressing information. This comprises the address, subnet mask, broadcast address, and default gateway. These values must be initialized either by way of DHCP or from NVRAM.

- MIB information. This is held within a MIB structure (defined in `\inet\net.h`), which is pointed to by the `n_mib` entry of the NET structure. The MIB information is used only if you have an SNMP agent running with your system. Values for the MIB structure must be initialized either when the interface is first prepared or when the `n_init()` function is called.

- Driver-specific local information. This is a generic **`void`** `*` pointer that can be assigned any value that you choose. You can also use it to point to a structure that gives more detail on your interface hardware, such as port addresses.

Note

In systems based on ARM processors, word data must be word aligned. This means that you must take care when coding Ethernet device drivers.

An Ethernet header contains 14 bytes of information. The sample Ethernet driver (in `\integrator\i8255x.c`) declares that this header is actually 16 bytes long in order to achieve 32-bit alignment of the other data structures within a packet, such as the IP header.

Specifying `n_lnh` to be 16 requires that the Ethernet driver processes incoming packets specially. It copies the first 14 bytes (the Ethernet header) into the packet data buffer, then skips two bytes in the buffer before copying in the remaining bytes from the Ethernet hardware. This allows both the Ethernet header and the IP or ARP header that follows it in the packet to be accessed starting on a 32-bit boundary.

2.2.1 n_close()

This function performs the tasks required to shutdown the device and its associated driver software prior to exiting the application. Any resources allocated to accommodate multiple packet types, for example, `0x0800` for IP and `0x0806` for ARP, must be released here. On embedded systems that start their devices at power up and do not shut them down, this function is not required.

Syntax

```
int n_close(int if_number)
```

where:

`if_number` is an index into the `nets[ ]` array of the interface to be closed.

Return value

Returns one of the following:

0 If successful.

ENP_Code If not successful (see *ENP_ error codes* on page A-2).

2.2.2 n_init()

This function prepares the device to send and receive packets. It is called during system startup after `prep_ifaces()` has been called, and before any of the other network interface functions are invoked.

Syntax

```
int n_init(int if_number)
```

where:

`if_number` Is an index into the `nets[ ]` array of the interface to be initialized.

Return value

Returns one of the following:

0 If successful.

ENP_Code If not successful (see *ENP_ error codes* on page A-2).

Usage

When this function returns, the device must be set up as follows:

- the network hardware must be ready to send and receive packets
- all required fields of the `nets[ ]` structure must be filled in
- the MIB-II structure of the interface must be filled in, as shown in Example 2-1 on page 2-19
- IP addressing information must be set before this function returns unless DHCP or BOOTP is to be used (see *Specifying IP addresses* on page 3-18).

This function typically includes hardware operations, such as initializing the device and enabling interrupts. It does not include setting protocol types. This is handled by `n_reg_type()`.

On returning from this function, it is safe for your hardware interrupt or receive functions to start queuing received packets in the `rvcdq` queue. Packets that are not IP or ARP are discarded by the stack.

The nets[ ] structure must be completely filled in when this function returns. The structure is defined in \inet\net.h. The work of filling this structure is shared between prep_ifaces() and n_init(). If all your nets[ ] structure setup was done in prep_ifaces() (see *prep_ifaces()* on page 2-11), there might be no additional work.

Example 2-1 shows sample code for setting up the MIB structure for a 10MB Ethernet interface. The n_mib field points to a structure that has already been statically allocated by the calling code. See RFC 1213 for detailed descriptions of the MIB fields.

Most of the MIB fields are used only for debugging and statistical information and are not critical unless your device is managed by SNMP.

You must make sure that nets[]->n_haddr points to a static buffer containing the MAC address before n_init() returns. The size of this address is determined by the media (6 bytes for Ethernet) and should be set in the nets[ ] structure member n_hal (hardware address length). You must also ensure that nets[]->n_type is set to the media type, for example, Ethernet.

Example 2-1

```
   np->n_lnh               = ETHHDR_SIZE;        /* 16 on ARM for alignment*/
   np->n_mtu               = 1500 + ETHHDR_SIZE; /* 1516 on ARM */
   np->n_hal               = 6;
   np->n_init              = i8255x_init;
   np->pkt_send            = i8255x_pkt_send;
   np->n_close             = raw_send
   ;
   np->n_stats             = NULL;

   np->n_local             = ( void * )dev;
#ifdef NET_STATS
   np->n_mib->ifOperStatus  = 2;                 /* interface is down */
   np->n_mib->ifAdminStatus = 2;                 /* interface is down */
   np->n_mib->ifLastChange  = cticks;
   np->n_mib->ifDescr       = (u_char *)i8255x_chip[revision];
   np->n_mib->ifIndex       = i8255x_next_net;
   np->n_mib->ifMtu         = ET_MAXLEN;
   np->n_mib->ifSpeed       = 10000000;
#endif
   np->n_haddr             = dev->mac_address;
   np->n_type              = ETHERNET;
```

See the sample Ethernet driver interface in \integrator\i8255x.c.

2.2.3 n_reg_type()

This function registers with lower level drivers to receive a MAC type, such as `0x0800` for IP and `0x0806` for ARP.

Syntax

```
int n_reg_type(unsigned short type, NET net)
```

where:

`type` Is the MAC type to be registered.

`net` Is a pointer to the NET structure corresponding to this interface.

Return value

Returns one of the following:

0 If successful.

ENP_Code If not successful (see *ENP_ error codes* on page A-2).

Usage

On most embedded systems with Ethernet, the ARM stack does not share the hardware with other network stacks, so no action is required. The stack gets all the packets and `n_reg_type()` can return an OK status without doing anything. However, you must make sure that all received packets are passed to the stack. On some driver subsystems, a type must be registered with the driver. On other drivers, an intermediate layer must be notified that the application is interested in the packets.

On PPP links, PPP processes the packets, so you do not have to modify `n_reg_type()`.

2.2.4 n_stats()

This is an optional function that you can use to display per-net statistics. You can also use it to log such statistics or return a pointer to a status block, depending on your requirements.

Syntax

```
void *n_stats(void *pio,  int if_number)
```

where:

`pio` Is a pointer to a GenericIO structure into which debug information is to be written.

`if_number` Is an index into the `nets[ ]` array of the interface for which statistics are to be dumped.

Return value

Optional.

Usage

This function is only used for debugging purposes.

2.2.5 pkt_send()

This function either sends the data in the passed PACKET structure or queues the PACKET structure for later transmission. If the MAC hardware is idle, the actual transmission of the packet must be started by this function, otherwise it must be scheduled to be sent later, usually by an *End Of Transmission* (EOT) interrupt from the hardware.

MAC headers for media, such as Ethernet or Token Ring, are placed at the head of the buffer passed by the calling function. Some drivers might have to access, strip, or modify the MAC header if they are layered on top of complex lower layers.

Syntax

```
int pkt_send(PACKET pkt)
```

where:

`pkt` Is the PACKET structure containing the frame to send.

Return value

Returns one of the following:

0 If successful.

ENP_code If not successful (see *ENP_ error codes* on page A-2).

Usage

The PACKET structure is defined in the file `\inet\netbuf.h`. All the information needed to send the packet is filled in before this call is made. The important fields are:

`pkt->nb_prot` Pointer to data to send.

`pkt->nb_plen` Length of data to send.

`pkt->net` `nets[ ]` structure for posting statistics.

The hardware driver must send `nb_plen` bytes, starting at `nb_prot`. When all the bytes are sent, the PACKET structure must be returned to the free queue by a call to `pk_free()`, which can be called at interrupt time. Do not free the packet before it has been successfully sent by the hardware, because it can then be reused (and its buffer altered) by the IP stack.

The simplest way to implement this function is to block (busy-wait) until the data is sent. This allows fast prototyping of new drivers, but generally affects performance. The usual design is to:

1. Put the packet in an `awaiting_send` queue.
2. Check to see if the hardware is idle.
3. Call a `send_next_from_q()` function to dequeue the packet at the head of the send queue.
4. Begin sending the packet.

The EOT interrupt frees the packet that has just been sent and calls the `send_next_from_q()` function again. Moving all the `PACKETS` through the `awaiting_send` queue ensures that they are sent in FIFO order. This significantly improves TCP and application performance.

If your hardware (or a lower layer driver) does not have an EOT interrupt or any analogous mechanism, you may need to use the `raw_send()` alternative to this function.

Slow devices, such as serial links and hardware that DMAs data directly out of predefined memory areas, can copy the passed buffer into driver-managed memory buffers, free the `PACKET`, and return immediately. However, these devices must be prepared to be called with more packets before transmission is complete.

Interface transmit functions must also maintain system statistics about packet transmissions. These are kept in the `n_mib` structure attached to each `nets[ ]` entry. Exact definitions of all these counters are available in RFC 1213. At a minimum, you should maintain packet byte and error counts, because these can help you debug your product during development and isolate configuration problems in the field. It is recommended that you perform statistics keeping at EOT time, but statistics can be approximated in this call. Example 2-1 shows a generic example.

Example 2-1

```
/* compile statistics about completed transmit */
eth = (struct ethhdr *)pkt->nb_prot;        /* get ether header */
ifc = pkt->net;
if(send_status==SUCCESSFUL)                 /* send_status set by hardware EOT */
{
    if(eth->e_dst[0] & 0x01)                /* see if multicast bit is on */
        ifc->n_mib->ifOutNUcastPkts++;
    else
        ifc->n_mib->ifOutUcastPkts++;
```



```
    ifc->n_mib->ifOutOctets +=pkt->nb_plen;
}
else                                        /* error sending packet */
{
    ifc->n_mib->ifOutErrors++;
}
```

Because this function may not wait for the packet transmission to complete, depending on your implementation, you can return a 0 if the packet has been successfully queued for send, or the send is in progress. Error (nonzero) codes should only be returned if a distinct hardware failure is detected. There is no mechanism to report errors detected in previous packets or during the EOT interrupt.

2.2.6 raw_send()

This function transmits the data on the device corresponding to the `nets[ ]` entry passed to it. Any MAC header required is placed at the head of the buffer passed by the calling function. This function must not return until it has finished processing the data in the passed buffer, because the buffer may be reused (corrupting the data) immediately upon return.

Note

The `pkt_send()` function (*pkt_send()* on page 2-22) must be used instead of this function if there is an EOT interrupt available on the hardware. This function is designed for old packet driver type drivers that do not support EOT and might not be needed otherwise. The function you are not using (either `pkt_send()` or `raw_send()`) must be set to NULL in the `nets[ ]` structure.

Syntax

```
int raw_send(NET net, char *data, unsigned data_bytes)
```

where:

`net` Is the net structure on which to send it.

`data` Is a pointer to the data buffer to send.

`data_bytes` Is the number of bytes to send (length of data).

Return value

Returns one of the following:

0 If successful.

ENP_Code If not successful (see *ENP_ error codes* on page A-2).

Usage

Slow devices (such as serial links) and hardware that DMAs data directly out of predefined memory areas can copy the passed buffer into driver managed memory buffers and return immediately. However, these devices must be prepared to be called with more data before transmission is complete.

Interface transmit functions must also maintain system statistics about packet transmissions. These are kept in the `n_mib` structure attached to each `nets[ ]` entry. Exact definitions of all these counters are available in RFC 1213.

At a minimum, you must maintain packet byte and error counts, because these can aid greatly with debugging your product during development and isolating configuration problems in the field. It is recommended that you perform statistics keeping at EOT time, but these can be approximated in this call. Example 2-1 is a generic example.

Example 2-1

```
/* compile statistics about completed transmit */
eth = (struct ethhdr *)data;                /* get ether header */
if(send_status == SUCCESSFUL)               /* send_status read from hardware */
{
    if(eth->e_dst[0] & 0x01)                /* see if multicast bit is on */
        net->n_mib->ifOutNUcastPkts++;
    else
        net->n_mib->ifOutUcastPkts++;

    net_>n_mib->ifOutOctets +=databytes;
}
else                                        /* error sending packet*/
{
    net->n_mib->ifOutErrors++;
}
```


Chapter 3
DHCP Client Functions

DHCP is used for configuring a network interface using information stored on a remote server. This chapter describes the function calls used to request configuration information for an interface. It contains the following section:

3.1 DHCP client functions

All DHCP client functions are found in `inet\dhcpclnt.c`. Many of them also require `inet\dhcputil.c` to be added to your project. The DHCP client functions are:

- *dhc_init()*
- *dhc_discover()*
- *dhc_set_callback()* on page 3-3
- *dhc_halt()* on page 3-4
- *dhc_second()* on page 3-4.

3.1.1 dhc_init()

This function initializes the DHCP client. It must be called before attempting to use DHCP to configure any network interfaces. This function attempts to open a UDP connection to listen for incoming replies. After DHCP has been initialized, the application must call `dhc_second()` one time every second.

Syntax

```
int dhcinit(void)
```

Return value

Returns one of the following:

0 If successful.

ENP_ code If not successful (see *ENP_ error codes* on page A-2).

3.1.2 dhc_discover()

This function begins the process of configuring a network interface using DHCP. The process may take several seconds, and the DHCP client will keep retrying if the service does not respond. You must abandon the attempt using `dhc_halt()` (see *dhc_halt()* on page 3-4) if the interface is not configured within a reasonable period (such as 30 seconds).

Syntax

```
int dhc_discover(int net)
```

where:

`net` Is the index of the network interface to be configured.

Return value

Returns one of the following:

0 If successful.

ENP_ code If not successful (see *ENP_ error codes* on page A-2).

Usage

A simple way to check for completion is to set the IP address for the interface to `0.0.0.0` before starting, and waiting until it becomes nonzero, as illustrated in Example 3-1.

Example 3-1

```
#ifdef DHCP_CLIENT
    dhc_init();
    for(i = 0; i < MAXNETS; i++)
    {
        dprintf("Using DHCP to obtain IP address information for
interface %d\n", i );
        nets[i]->n_ipaddr = 0L;
        dhc_discover(i);
    }
    do {
        tk_yield();
        e = 0;
        for(i = 0; i < MAXNETS; i++)
        {
            if ( nets[i]->n_ipaddr )
                e++;
        }
    } while ( e != MAXNETS );
    dprintf("All interfaces are now configured.\n" );
#endif
```

As an alternative to polling the IP address, you can install a callback to signal completion. See *dhc_set_callback()* for details.

3.1.3 dhc_set_callback()

This function installs a callback to be used to signal that the configuration of an interface is complete. This allows a multi-threaded RTOS to use DHCP without the overhead of polling loops.

Syntax

```
void dhc_set_callback(int net, int(*routine)(int, int))
```

where:

`net` Is the index of the network interface of interest.

`routine` Is the function to call when configuration is complete. This receives the index of the interface and the current DHCP state of the interface.

Return value

None.

Usage

This function is used to inform the application that the network interface is now available for use. It typically does this by setting a volatile flag, or by signaling the RTOS thread(s) waiting for the interface. The callback routine is called several times per interface, each time with an updated state. The interface can be used when the state is DHCS_BOUND.

3.1.4 dhc_halt()

This function stops all DHCP activity on a network interface.

Syntax

```
void dhc_halt(int net)
```

where:

`net` Is the index of the network interface on which to abandon DHCP.

Return value

None.

3.1.5 dhc_second()

This function must be called by the application exactly once every second. It is used to handle retry timeouts and lease expiry.

Syntax

```
void dhc_second(void)
```

Return value

None.

Chapter 4
Sockets

This chapter documents the sockets layer. Sockets are an API, primarily used for TCP programming. It provides a functional reference for the socket subset supported by the ARM TCP/IP stack. For more general information on sockets programming, many books and tutorials are available, for example, *Internetworking with TCP/IP*.

This chapter contains the following sections:

4.1 ARM implementation of sockets

In the ARM implementation of sockets, function names start with `t_`, for example, `socket()` is `t_socket()`. The names have been changed so that existing embedded systems that use standard socket functions do not have a conflict at link time. By adding the appropriate definitions to your `tcpport.h` file, you can continue to use the original socket function names in your code.

Also in the ARM implementation, the UNIX `errno` mechanism has been replaced by an error holder attached to each socket structure. The error holder is assigned a value when an error occurs. When a socket call indicates failure, you can examine this member or call `t_errno(socket)` to find out what went wrong. Possible values for sockets errors are listed in `tcp\nptcp.h`. These errors are a subset of the standard Berkeley sockets errors and are documented in *Socket error codes* on page A-4.

4.2 Socket API reference

This section contains an alphabetical list of the socket functions supported by ARM TCP/IP. Most of these are defined in `tcp\sockcall.c`:

4.2.1 t_accept()

This function is used to accept a connection from a remote host.

Syntax

```
long t_accept(long socket, struct sockaddr *addr)
```

where:

socket	Is a socket created with t_socket(), bound to an address with t_bind(), and is waiting for connections after a t_listen().
addr	Is the returned IP address and port number of the connecting entity (as known to the communications layer).

Return value

Returns one of the following:

descriptor	A descriptor for the accepted socket if successful.
-1	If not successful. On failure, the t_accept() function sets an internal socket variable, errno, to one of the errors listed in ipport.h. The value of errno can be retrieved by a call to t_errno(socket).

Usage

The t_accept() function is used with connection-based socket types, currently with SOCK_STREAM.

This function extracts the first connection on the queue of pending connections, creates a new socket with the same properties as socket, and allocates a new socket descriptor for the socket.

If no pending connections are present on the queue and the socket is not marked as nonblocking, t_accept() blocks the caller until a connection is present. If the socket is marked as nonblocking and no pending connections are present on the queue, t_accept() returns -1 and sets the socket errno to EWOULDBLOCK.

The accepted socket is used to read data to and write data from the socket that connected to this one. It is not used to accept more connections. The original socket, socket, remains open for accepting further connections.

It is possible to t_select() a socket for the purposes of doing a t_accept() by selecting it for read.

4.2.2 t_bind()

This function assigns a name to an unnamed socket. When a socket is created with `t_socket()`, it exists in a name space (address family) but has no name assigned.

Syntax

```
int t_bind(long socket, struct sockaddr *name)
```

where:

`socket` Is the identifier of the unnamed socket to be bound.

`name` Is the IP address and port number to be assigned to `socket`.

Return value

Returns one of the following:

`0` If successful.

`-1` If not successful. The internal socket variable `errno` is set to one of the errors listed in `ipport.h`. The value of `errno` can be retrieved by a call to `t_errno(socket)`.

4.2.3 t_connect()

This function creates a socket connection.

Syntax

```
extern int t_connect(long socket, struct sockaddr *name)
```

where:

`socket` Is created by `t_socket()`. It is bound to an IP address and port number using `t_bind()`.

If the type, as determined when the socket was created using `t_socket()`, is `SOCK_DGRAM`, `t_connect()` specifies the peer with which the socket is to be associated. This is the address to which datagrams are sent and is the only address from which datagrams are received.

If the type is `SOCK_STREAM`, `t_connect()` attempts to make a connection to another socket.

`name` Is an address in the communications space of the remote socket. Each communications space interprets the `name` parameter in its own way.

Return value

Returns one of the following:

`0` If successful.

`-1` If not successful. The internal socket variable `errno` is set to one of the errors listed in `ipport.h`. The value of `errno` can be retrieved by a call to `t_errno(socket)`.

Usage

Generally, `SOCK_STREAM` sockets can successfully use `t_connect()` only once. `SOCK_DGRAM` sockets can use `t_connect()` multiple times to change their association. Datagram sockets may dissolve the association by connecting to an invalid address, such as a null address.

4.2.4 t_errno()

This function is used to retrieve the current value of the error flag associated with a socket. The error status is *not* reset by this call.

Syntax

```
int t_errno (long s)
```

where:

`s` Is a socket descriptor.

Return value

The current error value associated with this socket (see *Socket error codes* on page A-4).

4.2.5 t_getpeername()

This function returns the IP addressing information of the connected host.

Syntax

```
int t_getpeername(long socket, struct sockaddr *name)
```

where:

`socket` Is the socket on which addressing information for the remote, connected host is returned.

`name` Is the returned IP address and port information.

Return value

Returns one of the following:

`0` If successful.

`-1` If not successful. The internal socket variable `errno` is set to one of the errors listed in `ipport.h`. The value of `errno` can be retrieved by a call to `t_errno(socket)`.

4.2.6 t_getsockname()

This function returns the current name for the specified socket.

Syntax

```
int t_getsockname(long socket, struct sockaddr *name)
```

where:

`socket` Is the identifier of the socket to be named.

`name` Is the name of the specified socket. On return from `t_getsocketname()`, this parameter contains IP address and port number information for `socket`.

Return value

Returns one of the following:

`0` If successful.

`-1` If not successful. The internal socket variable `errno` is set to one of the errors listed in `ipport.h`. The value of `errno` can be retrieved by a call to `t_errno(socket)`.

4.2.7 t_getsockopt()

This function returns the options associated with a socket.

Syntax

```
int t_getsockopt(long socket, int optname, void *optval)
```

where:

`socket` Is the socket for which the option values are returned.

`optname` Is the name of the option to be examined. The options are discussed in more detail below.

`optval` Is a pointer to a location where the value of the requested option is to be stored. If no option value is to be returned, `optval` can be supplied as NULL. Other than `SO_LINGER`, most socket-level options take a pointer to an **`int`** parameter for `optval`.

Return value

Returns one of the following:

`0` If successful.

`-1` If not successful. The internal socket variable `errno` is set to one of the errors listed in `ipport.h`. The value of `errno` can be retrieved by a call to `t_errno(socket)`.

Options

The following options are available:

`SO_BIO` This returns the value 1 if the socket is currently set to be blocking.

`SO_BROADCAST`
If `optval` is nonzero, this indicates that datagrams may be broadcast on this socket.

`SO_DONTROUTE`
If `optval` is nonzero, this indicates that outgoing messages must bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

SO_ERROR This returns any pending error on the socket and clears the error status. It can be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

SO_KEEPALIVE
If optval is nonzero, this indicates that periodic transmission of messages on a connected socket is enabled.

SO_LINGER
This indicates the action taken when unsent messages are queued on socket and a t_socketclose() is performed. If the socket promises reliable delivery of data and SO_LINGER is set, the system blocks the process on the t_socketclose() attempt until it is able to transmit the data or until it decides it is unable to deliver the information. A timeout period, known as the linger interval, is specified in the t_setsockopt() call when SO_LINGER is requested.

If SO_LINGER is disabled and a t_socketclose() is issued, the system processes the close in a manner that allows the process to continue as quickly as possible.

SO_LINGER uses a pointer to a **struct** linger parameter, defined in socket.h, that specifies the desired state of the option and the linger interval.

SO_MAXMSG This returns the TCP maximum segment size (TCP_MSS) as defined in tcpport.h.

SO_MYADDR This returns the IP address of the primary network interface for this host.

SO_NBIO, SO_NONBLOCK
These return the value 1 if the socket is currently set to be nonblocking.

SO_OOBINLINE
If optval is nonzero, this option indicates that out-of-band data is placed in the normal data input queue as received. It is then accessible through t_recv() calls without the MSG_OOB flag. This is valid with protocols that support out-of-band data.

SO_REUSEADDR
If optval is nonzero, SO_REUSADDR indicates that the rules used in validating addresses supplied in a t_bind() call allow reuse of local addresses.

SO_RXDATA This returns the number of characters currently available for reading from the socket.

`SO_SNDBUF and SO_RCVBUF`
: These return the buffer sizes allocated for output and input buffers, respectively.

`SO_TYPE`
: This returns the type of the socket, such as `SOCK_STREAM`. It is useful for servers that inherit sockets on startup.

4.2.8 t_listen()

This function tells the socket library that `socket` is going to be used for accepting connections from other hosts.

Syntax

```
extern int t_listen(long socket, int backlog)
```

where:

`socket` Is the socket used for accepting connections.

`backlog` Defines the maximum length to which the queue of pending connections can grow. If a connection request arrives when the queue is full, the client receives the error message `ECONNREFUSED`.

Return value

Returns one of the following:

`0` If successful.

`-1` If not successful. The internal socket variable `errno` is set to one of the errors listed in `ipport.h`. The value of `errno` can be retrieved by a call to `t_errno(socket)`.

Usage

To accept connections:

1. A socket is created with `t_socket()`.
2. A backlog for incoming connections is specified with `t_listen()`.
3. The connections are then accepted with `t_accept()`.

The `t_listen()` call applies only to `SOCK_STREAM` sockets.

4.2.9 t_recv() and t_recvfrom()

These functions are used to receive messages from another socket.

Syntax

```
int t_recv(long socket, char *buffer, int length, int flags)

int t_recvfrom(long socket, char *buffer, int length, int flags
,              struct sockaddr *from)
```

where:

`socket` Is the identifier of the socket from which the messages are received. The socket is created with `t_socket()`.

`buffer` Is the received message. If a message is too long to fit in `buffer`, excess bytes may be discarded depending on the type of socket from which the message is received.

`length` Is the length of `buffer` in bytes.

`flags` Is formed by ORing zero or more of the following:

- `MSG_OOB` Reads any out-of-band data present on the socket, rather than the regular in-band data.
- `MSG_PEEK` Looks at the data present on the socket. The data is returned, but not consumed, so a subsequent receive operation will see the same data.

`from` Is either NULL, or points to a **`struct`** `sockaddr` that will be filled in by `t_recvfrom()` with the source address of the message.

Return value

Returns one of the following:

`number` The number of bytes received, if successful.

`-1` If not successful. The internal socket variable `errno` is set to one of the errors listed in `ipport.h`. The value of `errno` can be retrieved by a call to `t_errno(socket)`.

Usage

You can only use the `t_recv()` function on a connected socket (see `t_connect()`). The `t_recvfrom()` function can be used to receive data on a socket, whether it is in a connected state or not.

If no messages are available at the socket and the socket is blocking, the receive call waits for a message to arrive. If the socket is nonblocking (see *t_setsockopt()* on page 4-19), –1 is returned, with the external socket `errno` set to `EWOULDBLOCK`.

You can use the `t_select()` function to determine when more data arrives.

4.2.10 t_select()

This function examines the input/output descriptor sets (whose addresses are passed in `readfds`, `writefds`, and `exceptfds`) to see if some of their descriptors are ready for reading, ready for writing, or have an exceptional condition pending. On return, `t_select()` replaces the given descriptor sets with subsets consisting of the descriptors that are ready for the requested operation. The total number of ready descriptors in all the sets is returned.

Syntax

```
int t_select(fd_set *readfds, fd_set *writefds,
             fd_set *exceptfds, long timeout)
```

`readfds` Is the set of socket descriptors to be tested for available data to be read.

`writefds` Is the set of socket descriptors to be tested for available buffer space for write operations.

`exceptfds` Is the set of socket descriptors to be tested for pending exceptional conditions (if *out-of-band* data is available to be read).

`timeout` Is the wait interval in `cticks` clock ticks. If `timeout` is neither 0 nor –1, it specifies the maximum number of clock ticks (at `TPS` ticks per second) to wait for the selection to complete. If `timeout` is zero, `t_select()` modifies the descriptor sets to indicate which are ready for the requested operation, and returns immediately. If timeout is –1, `t_select()` blocks until at least one of the requested operations is ready, and there is no timeout.

Return value

Returns one of the following:

`+value` A positive value indicates the number of ready descriptors in the descriptor sets.

`0` Indicates that the time limit referred to by timeout has expired.

`-1` If not successful.

Usage

You can give the parameters `readfds`, `writefds`, and `exceptfds` as NULL pointers if no descriptors are of interest.

To determine if a call to `t_accept()` will return immediately, call `t_select()`, passing the socket as a member of the *readfds* set.

Note

Under rare circumstances, `t_select()` can indicate that a descriptor is ready for writing when, in fact, an attempt to write would block. This can happen if system resources necessary for a write are subsequently exhausted after the select has returned or are otherwise unavailable. If an application deems it critical that writes to a socket descriptor do not block, it should set the descriptor for nonblocking input/output using the `SO_NBIO` request to `t_setsockopt()`.

The descriptors are stored within the `fd_set` structures as opaque objects. The macros below are provided for manipulating such structures.

The behavior of these macros is undefined if an invalid descriptor value is passed.

`FD_ZERO` `FD_ZERO(fd_set *fdset)`

This macro initializes a descriptor set `fdset` to the null set.

`FD_SET` `FD_SET(long fd, fd_set *fdset)`

This macro includes a particular socket descriptor `fd` in `fdset`.

`FD_CLR` `FD_CLR(long fd, fd_set *fdset)`

This macro removes `fd` from `fdset`.

`FD_ISSET` `FD_ISSET(long fd, fd_set *fdset)`

This macro is nonzero if `fd` is a member of `fdset`, and zero otherwise.

4.2.11 t_send() and t_sendto()

These functions are used to transmit a message.

Syntax

```
int t_send(int s, char *msg, int len, int flags)

int t_sendto(int s, char *msg, int len, int flags,
             struct sockaddr *to)
```

where:

`s`	Is a socket descriptor created with `t_socket()`.
`msg`	Is a pointer to the data to be sent.
`len`	Is the number of bytes to be sent.
`flags`	Are flags that control how the data is to be sent (see below).
`to`	Is the destination to which the data is to be sent.

Return value

Returns one of the following:

`number`	The number of bytes received, if successful.
`-1`	If not successful. The internal socket variable `errno` is set to one of the errors listed in `ipport.h`. The value of `errno` can be retrieved by a call to `t_errno(socket)`.

Usage

You can only use the `t_send()` function when the socket is in a connected state. The `t_sendto()` function can be used at any time.

The `flags` parameter is formed from the bitwise OR of zero or more of the following:

`MSG_OOB`	Sends out-of-band data. Only `SOCK_STREAM` sockets support out-of-band data.
`MSG_DONTROUTE`	The `SO_DONTROUTE` option is turned on for the duration of the operation. It is used only by diagnostic or routing programs.

If the socket does not have enough buffer space available to hold the message being sent, the `t_send()` functions block, unless the socket has been placed in nonblocking input/output mode (see *t_setsockopt()* on page 4-19).

4.2.12 t_setsockopt()

This function sets the options associated with a socket.

Syntax

```
int t_setsockopt(long socket, int optname, char *optval)
```

where:

`socket` Is the identifier of the socket to be changed.

`optname` Is the name of the option to be set.

`optval` Is the value the option will be set to. The parameter should be nonzero to enable a boolean option, or zero if the option is to be disabled. Most socket-level options take an **`int`** parameter for `optval`.

Return value

Returns one of the following:

`0` If successful.

`-1` If not successful. The internal socket variable `errno` is set to one of the errors listed in `ipport.h`. The value of `errno` can be retrieved by a call to `t_errno(socket)`.

Options

The following options are available:

`SO_BIO` This sets the socket to blocking mode. Operations on the socket are blocked until completion.

`SO_BROADCAST` This boolean value requests permission to send broadcast datagrams on the socket. Broadcast was a privileged operation in earlier versions of the system.

`SO_CALLBACK` This registers a callback function for use with the TCP Zero-Copy API. See Chapter 6 *The TCP Zero-copy API* for more information.

`SO_DONTROUTE` This boolean value indicates that outgoing messages must bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

`SO_KEEPALIVE` This boolean value enables the periodic transmission of messages on a connected socket. If the connected party fails to respond to these messages, the connection is considered broken.

`SO_LINGER` This option controls the action taken when unsent messages are queued on socket and a `t_socketclose()` is performed.

If the socket promises reliable delivery of data and `SO_LINGER` is set, the system blocks the process on the `t_socketclose()` 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 the `t_setsockopt()` call when `SO_LINGER` is requested).

If `SO_LINGER` is disabled and a `t_socketclose()` is issued, the system will process the close in a manner that allows the process to continue as quickly as possible.

`SO_LINGER` uses a pointer to a **`struct`** `linger` parameter, defined in `socket.h`, that specifies the desired state of the option and the linger interval.

`SO_NBIO` This sets the socket to nonblocking mode. If further operations on the socket cannot complete immediately, they return –1 and set the socket `errno` variable to `EWOULDBLOCK`.

`SO_NOBLOCK` This boolean value enables or disables blocking mode on the socket. If `optval` is set to zero, the socket is in blocking mode. If `optval` is set to a nonzero value, the socket is in nonblocking mode.

`SO_OOBINLINE` With protocols that support out-of-band data, this boolean option requests that out-of-band data be placed in the normal data input queue as received. It will then be accessible with `t_recv()`.

`SO_REUSEADDR` This boolean value indicates that the rules used in validating addresses supplied in a `t_bind()` call must allow reuse of local addresses.

`SO_SNDBUF, SO_RCVBUF`
These are options to adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size can be increased for high-volume connections or can be decreased to limit the possible backlog of incoming data. The system places an absolute limit of 16KB on these values.

4.2.13 t_shutdown()

This function causes all or part of a full duplex connection on the socket associated with `socket` to be shut down.

Syntax

```
int t_shutdown(long socket, int how)
```

where:

`socket` Is the connection to be shut down.

`how` Is the method of shut down, specified as follows:

- **0** Further receives are disallowed
- **1** Further sends are disallowed
- **2** Further sends and receives are disallowed.

Return value

Returns one of the following:

`0` If successful.

`-1` If not successful. The internal socket variable, `errno`, is set to one of the errors listed in `ipport.h`. The value of `errno` can be retrieved by a call to `t_errno(socket)`.

4.2.14 t_socket()

This function creates an endpoint for communication and returns a descriptor.

Syntax

```
extern long t_socket (int domain, int type, int protocol)
```

where:

`domain` Specifies a communications domain within which communication takes place. It selects the protocol family that should be used. The protocol family is typically the same as the address family for the addresses supplied in later operations on the socket. These families are defined in the include file `tcp\socket.h`. The only currently understood format is `PF_INET` (ARPA Internet protocols).

`type` Specifies the semantics of communication. Currently allowed types are:

- `SOCK_STREAM` (TCP) Provides sequenced, reliable, two-way connection-based byte streams. An out-of-band data transmission mechanism can be supported.
- `SOCK_DGRAM` (UDP) Supports datagrams, connectionless, unreliable messages of a fixed (typically small) maximum length.

`protocol` Is the protocol to use. It must be set to zero for ARM IP.

Return value

Returns one of the following:

`descriptor` A descriptor for the accepted socket, if successful.

`-1` If not successful.

Usage

SOCK_STREAM sockets are full duplex byte streams, similar to UNIX pipes. A stream socket must be in a connected state before it can send or receive data.

A connection to another socket is created with a t_connect() call. When connected, data can be transferred using t_send() and t_recv(). When a session has been completed, t_socketclose() can be performed. Out-of-band data can also be transmitted and received, as described in the t_send() and t_recv() documentation. For more details, see *t_send() and t_sendto()* on page 4-18 and *t_recv() and t_recvfrom()* on page 4-14.

The communications protocols used to implement a SOCK_STREAM ensure that data is not lost or duplicated. If a piece of data (for which the peer protocol has buffer space) cannot be transmitted successfully within a reasonable length of time, the connection is considered broken. In this case, calls return with a value of -1 and ETIMEDOUT is written to the internal variable errno.

The protocols optionally keep sockets *warm* by forcing transmissions roughly every minute in the absence of other activity. An error is indicated if no response has been received on an otherwise idle connection for an extended period, for example, five minutes.

SOCK_DGRAM sockets allow datagrams to be sent to correspondents named in t_sendto() calls. Datagrams are generally received with t_recvfrom(), which returns the next datagram with its return address.

The operation of sockets is controlled by socket-level options. These options are defined in the file socket.h. The t_getsockopt() and t_setsockopt() functions are used to get and set options, respectively (see *t_getsockopt()* on page 4-10 and *t_setsockopt()* on page 4-19).

4.2.15 t_socketclose()

This function deletes a descriptor from the reference table. On the close of a socket, associated naming information and queued data are discarded.

Note

This is just `close()` on traditional sockets systems.

Syntax

```
int t_socketclose(long socket)
```

where:

`socket` Is the identifier of the socket to be closed.

Return value

Returns one of the following:

`0` If successful.

`-1` If not successful. The internal socket variable `errno` is set to one of the errors listed in `ipport.h`. The value of `errno` can be retrieved by a call to `t_errno(socket)`.

Chapter 5
Low-overhead UDP Functions

This chapter describes low-overhead UDP functions. The functions described here are low-overhead in that they require little processing time and offer a small memory footprint. It contains the following sections:

5.1 UDP functions

These calls to the UDP layer are provided for systems that do not need the overhead of sockets (see Chapter 4 *Sockets*). These routines impose a lower demand on CPU and system memory requirements than sockets. However, they do not offer the portability of sockets. Most of these functions are defined in `inet\udp.c`. The exceptions are documented.

The following sections describe the low-overhead UDP functions. They are as follows:

- *udp_alloc()*
- *udp_close()* on page 5-3
- *udp_free()* on page 5-4
- *udp_open()* on page 5-5
- *udp_send()* on page 5-6
- *udp_socket()* on page 5-7.

5.1.1 udp_alloc()

This returns a packet large enough for the UDP data. It works by adding the space needed for UDP, IP, and MAC headers to the `datalen` passed, and calling `pk_alloc()`. It also ensures that the FREEQ_RESID resource is locked around the call to `pk_alloc()`.

Syntax

```
PACKET udp_alloc(int datalen, int optlen)
```

where:

`datalen` Is the length of UDP data, not including the UDP header.

`optlen` Is the length of IP options, if any. This is typically 0.

Return value

Returns one of the following:

`PACKET` A pointer to a packet buffer.

`NULL` If a large-enough packet was not available.

When the packet has been successfully transmitted by the hardware, it must be released by calling `udp_free()`. This is usually done by the sending interface.

5.1.2 udp_close()

This function must be called by your application when it has finished with a UDP connection and is no longer interested in receiving UDP packets associated with the connection. This function is defined in `inet\udp_open.c`.

Syntax

```
void udp_close(UDPCONN con)
```

where:

`con` Is the UDP connection identifier returned by a previous call to `udp_open()`.

Return value

None.

5.1.3 udp_free()

This function returns a previously allocated packet to the free pool on the stack. It calls `pk_free()` and ensures that the FREEQ_RESID resource is locked around the access to the free packet pool.

Syntax

`void udp_free(`*`ptr`*`)`

where:

`ptr` Is a pointer to the `netbuf` structure previously allocated by `udp_alloc()`.

Return value

None.

5.1.4 udp_open()

This function is defined in `inet\udp_open.c` and creates a structure in the UDP layer to receive, and pass upwards, UDP packets that match the parameters passed. The foreign host, `fhost`, and port, `fport`, can be set to 0 as a wild card, which enables the reception of broadcast datagrams.

The callback handler function is called with a pointer to a received datagram and a copy of the `data` pointer which is passed to `udp_open()`. This can be any data the programmer requires, such as a pointer to another function, or a control structure to aid in demultiplexing the received UDP packet.

Syntax

```
UDPCONN udp_open(ip_addr fhost,
                 unsigned short fport,
                 unsigned short lport,
                 int (*handler)(PACKET, void *),
                 void *data)
```

where:

`fhost` Is the foreign host from which you will accept data. It must be set to 0 if listening for any host.

`fport` Is the foreign port number. It must be set to 0 if listening for datagrams from any foreign port.

`lport` Is the local port on which to receive data.

`handler` Is the UDP receive callback function.

`data` Is the data that is passed (along with the received UDP datagram) to the callback handler.

Return value

Returns one of the following:

`ID` A UDP connection identifier if successful. This handle must be passed to `udp_close()` when the connection is no longer required.

`NULL` If not successful.

5.1.5 udp_send()

This function sends a UDP datagram to the foreign host in `pkt->fhost`. Local and remote ports in the UDP header are set from the values passed.

Syntax

```
int udp_send(unsigned short fport,
             unsigned short lport,
             PACKET pkt)
```

where:

`fport` Is the target UDP port.

`lport` Is the local UDP port.

`pkt` Is the packet to send, with `nb_prot`, `nb_plen`, and `fhost` (members of the `PACKET` structure) set.

Return value

Returns one of the following:

`0` If successful.

`ENP_Code` If not successful (see *ENP_ error codes* on page A-2).

5.1.6 udp_socket()

This function is used to obtain a random port number that is suitable for use as the `lport` parameter in a call to `udp_open()`. The `udp_socket()` function avoids picking port numbers in the *reserved* range 0-1024, or in the range 1025-1199, which may be used for server applications.

Syntax

```
int udp_socket(void)
```

Return value

Returns a UDP port number. This number is the `lport` parameter that is suitable for passing to `udp_open()`.

Chapter 6
The TCP Zero-copy API

This chapter describes the TCP Zero-copy API, an optional extension to the Sockets layer. It contains the following sections:

6.1 About the TCP Zero-copy API

This section documents an optional extension to the Sockets layer, the TCP Zero-copy API. This extension is only present if the stack has been built with the `TCP_ZEROCOPY` package option defined in `ipport.h`.

The TCP Zero-copy API is intended to assist the development of higher-performance embedded network applications by allowing the application direct access to the TCP/IP stack packet buffers. This feature can be used to avoid the overhead of having the stack copy data between application-owned buffers and stack-owned buffers in `t_send()` and `t_recv()`, but the application has to fit its data into, and accept its data from, the stack buffers.

6.1.1 Content of the API

The TCP Zero-copy API comprises:

- two functions for allocating and freeing packet buffers
- a function for sending a packet buffer on an open socket
- an application-supplied callback function for accepting received packets
- an extension to the Sockets `t_setsockopt()` function for registration of the callback function.

The TCP Zero-copy API is small because it is simply an extension to the existing Sockets API that provides an alternate mechanism for sending and receiving data on a socket. The Sockets API is used for all other operations on the socket.

Allocating and freeing packet buffers

The two functions for allocating and freeing packet buffers are straightforward requests:

`tcp_pktalloc()` Allocates a packet buffer from the pool of packet buffers on the stack.

`tcp_pktfree()` Frees a packet buffer.

Applications using the TCP Zero-copy API are responsible for allocating packet buffers for use in sending data, as well as for freeing buffers that have been used to receive data and those that the application has allocated but decided not to use for sending data. As these packet buffers are a limited resource, it is important that applications free them promptly when they are no longer of use.

Sending data through an open socket

The function for sending data, `tcp_xout()`, sends a packet buffer of data using a socket.

If successful, it is considered to have consumed the supplied buffer and so the application does not have to free the buffer using `tcp_pktfree()`.

Callback function

Applications that use the TCP Zero-copy API for receiving data must include a callback function for acceptance of received packets, and must register the callback function with the socket using the `t_setsockopt()` Sockets function with the `SO_CALLBACK` option name. The callback function, once registered, receives not only received data packets, but also connection events that result in socket errors.

6.2 Sending data with the TCP Zero-copy API

This section describes the procedure for allocating a packet buffer and sending data, in the following sections:

- *Allocating a packet buffer*
- *Filling the allocated buffer with data* on page 6-5
- *Sending the packet* on page 6-5.

6.2.1 Allocating a packet buffer

The first step in using the TCP Zero-copy API to send data is to allocate a packet buffer from the stack using the `tcp_pktalloc()` function. This function takes a single argument (the maximum length of the data you intend to send in the buffer) and returns a PACKET, a pointer to a network buffer structure, as shown in Example 6-1.

Example 6-1 The tcp_pktalloc() function

```
PACKET pkt;                    /* pointer to netbuf structure for packet buffer */
int datalen;                   /* amount of data to send */

datalen = 512;                 /* should indicate amount of data to send */
pkt = tcp_pktalloc(datalen);
if (pkt == NULL)
{
    /* error, could not allocate packet buffer */
}
```

Note

This limits how much data that you can send in one call using the TCP Zero-copy API, as the data sent in one call to `tcp_xout()` must fit in a single packet buffer, with the TCP, IP, and lower-layer headers that the stack needs to add in order to send the packet.

The actual limit is determined by the big packet buffer size, `bigbufsiz`, less the `HDRSLEN` definition in `tcpport.h`. If you try to request a larger buffer than this, `tcp_pktalloc()` returns NULL to indicate that it cannot allocate a sufficiently-large buffer.

6.2.2 Filling the allocated buffer with data

Having allocated the packet buffer, you now fill it with the data to send. The function `tcp_pktalloc()` has initialized the returned PACKET and so `pkt->nb_prot` points to where you can start depositing data.

When you have filled the buffer, you must set `pkt->nb_plen` to the number of bytes of data that you have placed in the buffer.

6.2.3 Sending the packet

Finally, you send the packet by giving it back to the stack using the function `tcp_xout()`.

```
e = tcp_xout(s, pkt);
if (e < 0)
{
    tcp_pktfree(pkt);
}
```

This function sends the packet over TCP, or returns an error. If its return value is less than zero, it has not accepted the packet and the application must either free the packet or retain it for sending later.

6.3 Receiving data with the TCP Zero-copy API

This section describes how you write and register a callback function, in the sections:

- *Writing a callback function*
- *Registering the callback function* on page 6-7.

6.3.1 Writing a callback function

Using the TCP Zero-copy API for receiving data requires the application developer to write a callback function that the stack can use to inform the application of received data packets and other socket events. This function is expected to conform to the following prototype:

```
int rx_callback(struct socket * so, PACKET pkt, int code);
```

The stack calls this function when it has received a data packet or other event to report for a socket, where:

`so` Identifies the socket.

`pkt` Passes a pointer to the packet buffer (if there is a packet buffer).

If `pkt` is not NULL, it is a pointer to a packet buffer containing received data for the socket. `pkt->nb_prot` points to the start of the received data, and `pkt->nb_len` indicates the number of bytes of received data in this buffer.

`code` Passes an error event (if there is an error to report).

If `code` is not 0, it is a socket error indicating that an error or other event has occurred on the socket. Typical nonzero values are:

`ESHUTDOWN` The connected peer has closed its end of the connection and sends no more data.

`ECONNRESET` The connected peer has abruptly closed its end of the connection and neither sends nor receives more data.

Identifying which socket is in use

If the application is using the same callback function for several sockets, it can use `so` to identify the socket for which the callback has occurred. For example, the following code fragment walks a list of data structures to find one with a matching socket, and illustrates a way to compare the so argument with a socket returned by `t_socket()`.

```
for (ftps = ftplist; ftps; ftps = ftps->next)
    if(ftps->datasock == so)
        break;
```

Once the callback function has identified the socket, it must examine the `pkt` and `code` parameters, as these contain the information about the socket.

Returned values

If the callback function returns 0, it indicates that it has accepted responsibility for the packet buffer and returns it to the stack (using the `tcp_pktfree()` function) when it no longer requires the buffer. If the callback function returns any nonzero value, it indicates to the stack that it has not accepted responsibility for the packet buffer. The stack keeps the packet buffer in the queue and calls the callback function again at a later time.

Note

The callback function is called from the stack and is expected to return promptly. Some of the places where the stack calls the callback function require that the data structures on the stack remain consistent through the callback, so the callback function must not call back into the stack except to call `tcp_pktfree()`. (This restriction might be removed in a future release of the stack.)

6.3.2 Registering the callback function

The application must also inform the stack of the callback function. If the stack has been built with the `TCP_ZEROCOPY` option enabled, the `t_setsockopt()` function provides an additional socket option, `SO_CALLBACK`, which should be used for this purpose once the socket has been created. The following code fragment illustrates the use of this option to register a callback function named `rxupcall()` on the socket `sock`:

```
t_setsockopt(sock, SO_CALLBACK, (void *)rxupcall);
```

The function `t_setsockopt()` is described in *t_setsockopt()* on page 4-19.

6.4 TCP Zero-copy API reference

This section gives the syntax and description of the following functions:

- *tcp_pktalloc()*
- *tcp_pktfree()* on page 6-9
- *tcp_xout()* on page 6-9.

6.4.1 tcp_pktalloc()

This function allocates a packet buffer. It is a small wrapper around the internal `pk_alloc()` function that provides the necessary synchronization and calculation of header length.

Syntax

```
PACKET tcp_pktalloc(int datalen);
```

where:

`datalen` Is the length of TCP data (not including the TCP header).

Usage

This function allocates a packet buffer large enough to hold `datalen` bytes of TCP data, plus TCP, IP and MAC headers.

This function must be called to allocate a buffer for sending data via `tcp_xout()`. It returns the allocated packet buffer with its `pkt->nb_prot` field set to where the application must deposit the data to be sent.

Return value

Returns one of the following:

`pointer` A pointer to **`struct`** `netbuf` if the allocation was successful.

`NULL` If a big enough packet was not available.

6.4.2 tcp_pktfree()

This function frees a packet buffer allocated by `tcp_pktalloc()`. It is a small wrapper around the internal `pk_free()` function that provides necessary synchronization.

Syntax

```
void tcp_pktfree(PACKET pkt);
```

where:

`pkt` Is a pointer to packet to be freed.

Return value

No return value.

6.4.3 tcp_xout()

This function sends a packet buffer on a socket.

Syntax

```
int tcp_xout(long s, PACKET pkt);
```

where:

`s` Is the socket on which the packet is to be sent.

`pkt` Is a pointer to packet to be sent.

Usage

The packet buffer must be initialized with `pkt->nb_prot` pointing to the start of the application data to be sent (this was set by `tcp_pktalloc()`), and with `pkt->nb_plen` set to the number of bytes of data to be sent.

Return values

This function returns an integer indicating the success or failure of the function:

`0` Indicates that the packet was sent successfully.

`<0` Indicates that the packet was not accepted by the stack. The application must re-send the packet using a call to `tcp_xout()`, or free the packet using `tcp_pktfree()`.

`>0` Indicates that the packet has been accepted and queued on the socket but has not yet been transmitted.

Chapter 7
ARM-specific Functions

The sample sources provided as part of the TCP/IP stack contain several functions that are specific to the ARM environment. This chapter describes those files and the functions they contain. It contains the following sections:

7.1 ARM directories

The ARM-specific files are contained within the following directories:

`\armthumb`	Contains functions specific to ARM or Thumb processors.
`\integrator`	Contains functions specific to the ARM Integrator/AP development board.
`\uHAL`	Contains libraries and header files from the ARM Firmware Suite.

7.1.1 ARM-specific routines

The `\armthumb` directory contains files that provide functionality common to all ARM processors and development platforms, and contains the following files:

`armthumb.h`	C declarations for the functions implemented in this directory.
`asmacros.h`	Assembler macros used by files in this directory.
`cksum.s`	Optimized ARM assembler implementation of the checksum function, cksum().
`dtrap.s`	Stub function that can be used to trap to the debugger.
`lswap.s`	Optimized ARM assembler implementations of 32-bit and 16-bit endian swap routines.

armthumb.h

This C header file contains function declarations for the following functions implemented by other files in this directory:

```
extern unsigned short cksum( unsigned short *, unsigned );
extern void dtrap( void );
extern unsigned lswap( unsigned );
extern unsigned short bswap( unsigned short );
```

asmacros.h

This assembler header file contains macro definitions used by assembler files in the `\armthumb` and `\integrator` directories. The macros allow you to code assembler functions to handle interworked and non-interworked operation cleanly.

For interworking in ADS

Invoke armasm with `-PD "INTER SETA 2"` on the command line.

For interworking in SDT

Invoke armasm with `-PD "INTER SETA 1"` on the command line.

No interworking

In either ADS or SDT, invoke armasm with `-PD "INTER SETA 0"`, or omit the command-line predefine completely.

Of the macros defined in `asmacros.h`, the following are used by the stack:

```
RETMOV <register>, <cc> to simulate MOV<cc> pc, <register>
RETLDM <list>, <cc>to simulate LDM<cc>FD sp!, {<list>, pc}
```

cksum.s

The file `cksum.s` contains optimized ARM and Thumb versions of the `cksum()` function. IP, UDP, TCP, and PPP use this function to calculate the checksum of a block of data. The algorithm implemented is described by RFC 1071. A C version of this function is also provided in the `inet` directory, but this must only be used if you suspect that there is some problem with these optimized versions.

You can optimize the ARM implementation of this function by setting the `NREG` variable within this file to a value between one and eight. `NREG` controls the number of registers used to perform the checksumming operation. The default value of five has been selected through a benchmarking process. The ARMulator is used to emulate an ARM7TDMI core without caching, while performing TCP and UDP transfers of varying block sizes using the Loopback example program. You might want to run performance tests on your system to determine the best value to use, especially if the processor you are using has a data cache.

dtrap.s

The function `dtrap()`, implemented in this file, does nothing. If the RTOS you are using supports a *trap to debugger* call, you can call that from this function. It is possible to use an `.obey` file containing the command `break @_dtrap` to cause the debugger to halt execution of the system when `dtrap()` is called.

You do not need to include `dtrap.s` in production code because you have eliminated all possible causes of `dtrap()` before production.

lswap.s

You must include `lswap.s` only if you are implementing a little-endian system.

When using the *Internet Protocol* (IP), all protocol data larger than a single byte is transmitted on the network in network byte order (big-endian mode). If your system is operating in little-endian mode, all 16-bit and 32-bit quantities that are sent or received must be byte-swapped from network byte order to little-endian byte order. Traditionally, four macros are defined:

`htonl(l)` Converts a 32-bit value from host byte order to network byte order.

`ntohl(l)` Converts a 32-bit value from network byte order to host byte order.

`htons(s)` Converts a 16-bit value from host byte order to network byte order.

`ntohs(s)` Converts a 16-bit value from network byte order to host byte order.

For 32-bit quantities, you define `htonl()` and `ntohl()` to use the `lswap()` function implemented in the `lswap.s` file. This is an optimized routine for performing endian-swap operations on 32-bit values.

For 16-bit quantities, you either use the `bswap()` function implemented in the `lswap.s` file, or you can use the following macros if you are optimizing for speed rather than code size, and are using ARM state rather than Thumb state:

```
#define htons(s) ((u_short)(((u_short)(s)>>8) | ((u_short)(s)<<8)))
#define ntohs(s) htons(s)
```

7.1.2 Integrator/AP-specific routines

The `\integrator` directory contains files used to provide support for the Integrator/AP development platform fitted with an ARM Integrator/CM7TDMI core module. It contains the following files:

`clock.c` Provides the basic timer.

`crit.c` Provides critical section protection.

`i8255x.c` Is the Ethernet device driver for PCI NICs based on Intel 82557/82558/82559.

`lowlevel.s` Are the Assembler routines used by other files in this section.

`initboard.c` Contains the start-up routine that calls all of the other modules to initialize them.

`uartio.c` Is an interrupt-driven UART driver with hardware flow control and circular buffers.

clock.c

This file contains three functions, `clock_init()`, `clock_c()`, and `ticker()` that manage timer interrupts used solely to keep track of elapsed time:

`clock_init()`	Initializes one of the µHAL timers to generate a regular clock tick interrupt. The values used to program the timer are calculated from the TPS value defined in `ipport.h`.
`clock_c()`	Reverses the actions performed by `clock_init()` by freeing the µHAL timer. This stops the timer running, and disables the interrupts generated by it.
`clock_c()`	Called in timer interrupt context to increment `cticks` `TPS` times every second.

crit.c

This file contains implementations of `ENTER_CRIT_SECTION()` and `EXIT_CRIT_SECTION()`.

This particular implementation:

- disables all interrupts when `ENTER_CRIT_SECTION()` is called
- keeps track of nested critical sections
- re-enables interrupts when `EXIT_CRIT_SECTION()` is called on the outermost critical section.

It also contains code to check that the critical sections are balanced. That is, `EXIT_CRIT_SECTION()` is called with the same pointer value that the corresponding `ENTER_CRIT_SECTION()` was called with.

`ENTER_CRIT_SECTION()` and `EXIT_CRIT_SECTION()` use assembler routines defined in `lowlevel.s` to test, set, and clear the interrupt disable bits in the ARM *Current Processor Status Register* (CPSR). Your implementation of `ENTER_CRIT_SECTION()` and `EXIT_CRIT_SECTION()` might have to use a different technique, perhaps only disabling interrupts from the network interface hardware, or disabling scheduler pre-emption in an RTOS environment.

i8255x.c

This file contains a device driver for the Intel 82557, 82558, and 82559 PCI *Network Interface Controllers* (NICs). This device is a PCI bus master device that can manage complex buffering schemes, scatter and gather DMA, media management, and many other features.

This driver implements a subset of the full functionality of the 82559 device so that it works with previous versions of the chip, such as the 82558 and 82557. The 82559 device is used on the Intel PRO/100+ Management Adapter.

The driver uses the PCI and µHAL libraries from the ARM Firmware Suite 1.1 to manage the low-level interface to the card. See *ARM Firmware Suite* on page 7-8 for information on using other variants of these libraries.

lowlevel.s

This file implements some low-level routines used by some of the device drivers in this directory:

`unsigned test_and_set_ibit( void );`

Sets the interrupt disable bit and returns the previous state.

`unsigned set_ibit( unsigned flag );`

Sets the interrupt disable bit if `flag` is true, and clears it otherwise.

`void nano( unsigned ns );`

Delays for `ns` nanoseconds.

`unsigned mrc_15( unsigned cpreg );`

Returns the current value of co-processor 15 register 2 (if `cpreg` equals 2) or register 1 (otherwise).

`void mcr_15( unsigned value );`

Writes `value` to co-processor 15 register 1.

initboard.c

This file implements the board-level startup sequence. The `initboard.c` file contains routines you can use to alter these settings, set the bus mode of the processor being used, and reprogram the MMU/MPU.

```
unsigned int InitialiseBoard( void );
```

Your `main()` routine must call `InitialiseBoard()` before doing anything else. `InitialiseBoard()` sets up the uHAL and PCI library environments, and initializes the UART drivers.

uartio.c

This file contains a device driver for the PrimeCell UARTs used on the Integrator/AP motherboard that:

- is interrupt-driven
- implements hardware flow control
- uses large circular buffers.

`uartio.c` has implementations of `dputchar()`, `kbhit()`, and `getch()` that are used by the menus system and for debug:

```
int  uart_getc( int unit );

void uart_putc( int unit, int ch );

int  uart_ready( int unit );

int  uart_stats(void *pio, int unit);

void uart_set_dtr(int unit, int state);

int uart_get_dcd(int unit);
```

The file also implements the UART interface required by the ARM PPP software package which is available separately. The UART interface is described in the *Porting PPP Programmer's Guide*.

7.2 ARM Firmware Suite

The \uHAL directory contains µHAL and PCI libraries and header files taken directly from the *ARM Firmware Suite* (AFS) Version 1.1 CD-ROM. Using µHAL allows you to port the stack with ease to other development platforms that support µHAL with.

The libraries shipped with ARM TCP/IP are the IntegratorT variants, that work with the following core modules:

- Integrator/CM7TDMI
- Integrator/CM720T
- Integrator/CM740T
- Integrator/CM920T
- Integrator/CM940T.

This supports ARM/Thumb interworking.

Note

The IntegratorT variant does not support the memory management and protection units available on the CM720T/740T/920T/940T core modules. If you want to use this, or any other functionality provided by µHAL for these core modules, you have to copy the appropriate files from the AFS CD-ROM.

7.2.1 Example

As an example, the libraries for the Integrator/CM740T are in the following locations on the CD-ROM:

```
\common\images\Integrator740T\uHAL\Build\Integrator740T.b\
    semihosted\uHALlibrary.a

\common\images\Integrator740T\PCI\Build\Integrator740T.b\
    semihosted\PCIlib.a
```

The header files in the µHAL directory are copied from the following locations on the CD-ROM:

```
\{windows|unix}\source\all\uHAL\h

\{windows|unix}\source\all\uHAL\Processors

\{windows|unix}\source\all\PCI\Sources

\{windows|unix}\source\all\uHAL\Boards\INTEGRATOR
```


Chapter 8
Miscellaneous Library Functions

This chapter describes the assortment of functions that are found in the `\misclib` directory. These functions perform a variety of tasks that are used by the example programs, and by the TCP/IP stack. You might not require all, or any, of these functions in your final system. These functions enable you to perform a sample port, but they do not comprise part of the supported product. It contains the following sections:

8.1 Description of misclib files

This section lists the files in `misclib` and gives an overview of their functions:

- *app_ping.c*
- *in_utils.c*
- *memman.c*
- *menus.c, menulib.c, and nrmenus.c* on page 8-3
- *nextcarg.c* on page 8-3
- *nvparms.c* on page 8-3
- *parseip.c* on page 8-3
- *reshost.c* on page 8-4
- *strilib.c* on page 8-4
- *strlib.c* on page 8-4
- *tcp_echo.c* on page 8-4
- *timeouts.c* on page 8-4
- *testmenu.c* on page 8-5
- *ttyio.c* on page 8-5
- *udp_echo.c* on page 8-5
- *userpass.c* on page 8-5.

8.1.1 app_ping.c

The functions within the `app_ping.c` file implement an interface to the *ping* facilities available in the `\inet` directory for the menus subsystem.

8.1.2 in_utils.c

This file contains an assortment of general utility functions. They are described in *in_utils.c* on page 8-6.

8.1.3 memman.c

The ARM networking software uses the two functions `npalloc()` and `npfree()` to make dynamic memory allocations. In systems that include the standard C library, these can be mapped directly onto `calloc()` and `free()` using defined macros in the `ipport.h` file.

If you are experiencing memory allocation problems and suspect that memory blocks are being referenced after they have been freed, or that data is being written beyond the end of the allocated area, you may be able to use the `npalloc()` and `npfree()` functions implemented in `memman.c` to help with debugging. If you are using these functions, you may also use the diagnostic function `blocklist()` which uses `dprintf()` to print a list of the memory blocks currently in use.

`memman.c` also contains a function, `check_memory()`, that you can call at any point in your code. It checks that all of the structures used by the dynamic memory allocation routines in `memman.c` are still intact and have not been accidentally overwritten. Typically, you would place calls to `check_memory()` before and after a section of code that you suspect is corrupting the dynamic memory heap.

8.1.4 menus.c, menulib.c, and nrmenus.c

These three files contain functions that implement a menu system that can be readily extended as new modules are added to a project. The menu system is intended to run with your application code, and allows you to exercise different areas of the network protocols. The menus can be accessed either by way of the standard input and output channels, or by way of a TELNET socket or other GenericIO channel.

8.1.5 nextcarg.c

This file contains only one function, `nextcarg()`, that splits a comma-delimited string into its components. It is described in *nextcarg.c* on page 8-17.

8.1.6 nvparms.c

The `nvparms.c` file contains a large number of routines used to read and parse a configuration file. This configuration file can either reside in nonvolatile Flash memory, or in a file on the native file system. If you choose to store files in flash memory, you must define `INCLUDE_FLASHFS` in your `ipport.h` file and provide suitable definitions to `nv_functions` in `nvfsio.h`. If `INCLUDE_FLASHFS` has not been defined, the standard C library functions, such as `fopen()`, `fgets()`, and `fclose()`, are used.

8.1.7 parseip.c

The `parseip.c` file implements only one function, `parse_ipad()`, used to parse a string containing a *dotted-quad* IP address (for example, `192.168.117.43`) and return it as an `ip_addr` value. It is described in *parseip.c* on page 8-18.

8.1.8 reshost.c

The reshost.c file implements only one function, in_reshost(), used to resolve a host name into an IP address. It is described in *reshost.c* on page 8-19.

8.1.9 strilib.c

There are three functions, stricmp(), strnicmp(), and stristr(), that are implemented in strilib.c. They each perform case-independent string comparisons. They behave just like strcmp(), strncmp(), and strstr(), taking the same arguments and returning the same results, with the exception that each character of each string is converted to lowercase before being compared. The strings passed to these functions are *not* modified.

8.1.10 strlib.c

The strlib.c file contains implementations of the following standard C string functions:

- strcat()
- strchr()
- strcmp()
- strcpy()
- strlen()
- strncmp()
- strncpy()
- strstr().

If your C libraries do not include these functions, they may be included from the strlib.c file by defining the macro INICHE_LIBS in your ipport.h file, and configuring which particular functions you require in the in_utils.h file.

8.1.11 tcp_echo.c

The functions within the tcp_echo.c file implement a *menus* interface to the TCP echo mechanism, which is useful for testing purposes.

8.1.12 timeouts.c

This file contains a single function, inet_timer(), that should be called at least twice per second if you are using a superloop system. It is described in *timeouts.c* on page 8-21.

8.1.13 testmenu.c

This file adds three commands to the diagnostic menus system to allow you to send a large number of ICMP ECHO requests or ARP requests to a target host. It is described in *testmenu.c* on page 8-23.

8.1.14 ttyio.c

This file contains an implementation of the `dprintf()` function that can be used to send debug output by way of a UART driver. The `dprintf()` function supports the following formatting characters:

%x unsigned hex
%d signed decimal
%u unsigned decimal
%c character
%s null-terminated string
%p unsigned pointer (same as %x)
%lx unsigned long hex
%ld signed long decimal
%lu unsigned long decimal.

If this file is compiled with `FIELDWIDTH` defined, `%[[-]w][.][p]f` formats are understood, where `w` is the minimum field width and `p` is the precision, indicating the minimum number of digits to be printed. If `w` is negative, the field is left adjusted. If `w` starts with a leading zero, the field is padded using zeros instead of spaces.

This function uses `dputchar()` to perform output.

8.1.15 udp_echo.c

The functions within the `udp_echo.c` file implement a menus interface to the UDP echo mechanism, which is useful for testing purposes.

8.1.16 userpass.c

The `userpass.c` file contains code to implement a simple user/password database. Its functions are described in *userpass.c* on page 8-24.

8.2 in_utils.c

This file contains an assortment of general utility functions. They are described in this section. The functions are:

8.2.1 con_page()

This function implements a simple `more` facility.

Syntax

```
int con_page(void *vio, int lines)
```

where:

`vio` Points to a generic I/O structure (see *ns_printf()* on page 8-10).

`lines` Is a counter containing the number of lines printed so far.

Return value

Returns one of the following:

`0` If more output should be produced.

`1` If the user indicates that no more output is wanted.

Usage

The `con_page()` function implements a simple `more` facility that waits for a key press from the user if more than a screen of information has been displayed. Normally, `0` is returned. The value `1` is returned if the Escape key is pressed as a response to the `press any key for more` prompt, indicating that the user does not wish to see more output.

8.2.2 hexdump()

This function is used to display an area of memory, usually for debugging purposes.

Syntax

```
void hexdump(void *pio,  void * buffer, unsigned len)
```

where:

`pio` Points to a generic I/O structure (see *ns_printf()* on page 8-10).

`buffer` Points to the memory area to be displayed.

`len` Is the number of bytes to be displayed.

Return value

None.

Usage

The `hexdump()` function displays `len` bytes of data, starting from the address `buffer`, using the `dprintf()` routine.

8.2.3 nextarg()

This function is used to parse a string into separate arguments.

Syntax

```
char *nextarg(char *argp)
```

where:

`argp` Points to the string of arguments to be processed.

Return value

Returns a pointer to the *next* argument in `argp`, or to the terminating null character if there are no more arguments.

Usage

This function returns a pointer to the next argument in the string passed. Arguments are considered to be printable ASCII sequences of characters delimited by spaces. If there are no more arguments present within the string passed, this function returns a pointer to the null character that terminates the string. For example, calling `nextarg()` with the string `one two three` returns a pointer to the letter `t` of the word `two`. Calling `nextarg()` with that pointer returns a pointer to the letter `t` of the word `three`, and calling `nextarg()` once more returns a pointer to the terminating null character of the string.

8.2.4 ns_printf()

This function is used to report network statistics.

Syntax

```
int ns_printf(void *vio, char *format, …)
```

where:

`vio` Is a generic input/output pointer.

`format` Is a format string like `printf()`.

… Is a list of arguments, as described by `format`.

Return value

Returns one of the following:

`chars` is the number of characters printed, if successful.

`value` negative value, if not successful.

Usage

The `ns_printf()` function is used by the various network statistics printing routines within the stack to report current counter values. The `vio` argument is a pointer to a GenericIO structure, which can be used to direct output to different streams, such as the debug console or a TELNET socket.

8.2.5 panic()

This function is used to indicate a nonrecoverable fault within the system.

Syntax

```
void panic(char *msg)
```

where:

`msg` Is an informative message indicating the nature of the problem.

Return value

None.

Usage

This function uses `dprintf()` to print the message string passed, attempts to trap to the debugger using `dtrap()`, and then halts the system by calling `netexit()`. You must modify this code if you want your system to restart (warm boot) after such a failure.

8.2.6 print_eth()

This function is used to format an Ethernet address as an ASCII string for printing.

Syntax

```
char *print_eth(char *addr, char spacer)
```

where:

`addr` Points to the 6-byte Ethernet address.

`spacer` Is the character to be used between each octet.

Return value

Returns a pointer to a statically allocated buffer containing a null-terminated ASCII string that represents the Ethernet address pointed to by `addr`.

Usage

This function is useful for formatting Ethernet MAC addresses. It returns a pointer to a static buffer containing the MAC address represented as a string of six 2-digit hexadecimal numbers. Each number is separated from the next by the character `spacer`. The complete string of six numbers is null-terminated. If `spacer` is passed as `\0`, no spacing characters are inserted into the output string and the resulting string is 12 characters long. If `spacer` is nonzero, the resulting string is 17 characters long.

8.2.7 print_ipad()

This function is used to format an IP address as an ASCII string for printing.

Syntax

```
char *print_ipad(unsigned long ipaddr)
```

where:

`ipaddr` Is the network address to be printed, in network byte order.

Return value

Returns a pointer to a statically allocated buffer containing a null-terminated ASCII string that represents the Internet address passed in `ipaddr`.

Usage

This function returns a pointer to a static buffer that contains the dotted quad notation for the IP address passed. The IP address is expected to be in network byte order (big-endian).

8.2.8 print_uptime()

This function is used to format a time value as an ASCII string for printing.

Syntax

```
char *print_uptime(unsigned long timetick)
```

where:

`timetick` Is the number of ticks to be translated to an uptime string.

Return value

Returns a pointer to a statically allocated buffer containing a null-terminated ASCII string that represents the time value passed in `timetick`.

Usage

This function takes a time value, `timetick`, in centiseconds and returns a pointer to a static buffer containing an uptime string that indicates the number of days, hours, minutes, and seconds represented.

8.2.9 std_in()

This function inputs a character from the standard input channel.

Syntax

```
int std_in(long s)
```

where:

`s` Is the index of the input device (unused by `std_in()`).

Return value

Returns one of the following:

`0` If no character is available from the standard input channel.

`char` The character typed, if a character is available.

Usage

The `std_in()` function is used with a GenericIO structure to allow the menus system to receive input from the standard input channel.

8.2.10 std_out()

This function outputs characters to the standard output channel.

Syntax

```
int std_out(long s, char *buf, int len)
```

where:

`s` Is the index of the output device (unused by `std_out()`).

`buf` Is a pointer to the data to be printer.

`len` Is the number of characters to output from `buf`.

Return value

Returns the number of characters actually printed.

Usage

The `std_out()` function is used with a GenericIO structure to allow the `ns_printf()` function to print network statistics to the standard output channel.

8.2.11 sysuptime()

This function returns the age of the system.

Syntax

```
unsigned long sysuptime(void)
```

Return value

Returns the number of centiseconds since the clock started counting.

Usage

The `sysuptime()` function returns the time since some arbitrary epoch. This epoch is usually the moment when the clock driver was initialized at boot time. The time period is expressed in hundredths of a second.

8.2.12 uslash()

This function is used to translate DOS-style path separator characters (\) into UNIX-style separator characters (/).

Syntax

```
char *uslash(char *path)
```

where:

`path` Is the string to be translated.

Return value

Returns the string pointed to by `path`, with every occurrence of the \ character replaced with /.

Usage

The string is modified in place, and therefore must not be a constant (read-only) string.

8.3 nextcarg.c

The `nextcarg.c` file contains only one function, `nextcarg()`.

8.3.1 nextcarg()

This function is used to split a comma-delimited string into its components.

Syntax

```
char *nextcarg(char *arg)
```

where:

`arg` Is a pointer to a null-terminated string containing a comma-delimited argument list.

Return value

Returns one of the following:

`next` The next argument from `arg` if there is an argument to be returned.

`NULL` If there are no more arguments.

8.4 parseip.c

The `parseip.c` file implements only one function, `parse_ipad()`.

8.4.1 parseip()

This function is used to parse a string containing a *dotted-quad* IP address (for example, `192.168.117.43`) and return it as an `ip_addr` value.

Syntax

```
char *parse_ipad(ip_addr *ipout, unsigned *sbits,
                 char *stringin)
```

where:

`ipout` Is a pointer to an `ip_addr` that will contain the return value.

`sbits` Is a pointer to a location that will contain the number of subnet bits corresponding to the class of the network address passed.

`stringin` Is a pointer to the string to be parsed.

Return value

Returns one of the following:

`NULL` If successful, that is, if the string passed was fully parsed.

`pointer` A pointer to a string describing the problem if not successful, that is, if the string passed could not be fully parsed.

Usage

The `parse_ipad()` function understands that network addresses with zeroes in them may be abbreviated. For example, `127.1` is expanded to the address `127.0.0.1`. The `parse_ipad()` function also fills in the number of subnet bits corresponding to the class of the network address in the location pointed to by `sbits`. That is, `*sbits` will be set to 8 if the address is class A, 16 for class B, and 24 for a class C.

8.5 reshost.c

The reshost.c file implements only one function, in_reshost().

8.5.1 in_reshost()

This function is used to resolve a host name into an IP address.

Syntax

```
int in_reshost(char *host, ip_addr *address, int flags)
```

where:

host — Contains the host-name string to be resolved.

address — Points to the location where the resolved IP address is to be stored.

flags — Are flags to control how in_reshost() operates (see *Usage* below).

Return value

Returns one of the following:

0 — If successful.

ENP_Code — If not successful (see *ENP_ error codes* on page A-2).

Usage

The in_reshost() function is called with a string containing the host name to be resolved, either in dotted-quad notation (for example, 192.168.117.43), or as a *fully-qualified domain name* (for example, myhost.mydomain.com). The in_reshost() function will attempt to parse the address, first as a dotted-quad address, and then using *Domain Name System* (DNS) lookup (if configured into the system), and will fill in the ip_addr pointed to by address with the IP address of the host. The flags value is used to control how the lookup is performed, and consists of the following flags, ORed together:

RH_VERBOSE — Prints debugging/progress information about the request.

RH_BLOCK — Blocks until the address has been resolved, or until an error has been detected.

IF RH_BLOCK is *not* specified, and DNS resolution is required, this routine returns immediately, having sent the request to the DNS server. It is important that the calling routine should zero the location pointed to by address before calling in_reshost(), and should then poll this function until the value pointed to by literal becomes nonzero.

Note

Trying to resolve a local hostname, such as ahost, without a qualifying domain name, will almost certainly fail. For domain-name resolution to be successful, the fully-qualified domain name is required.

8.6 timeouts.c

This file contains a single function, `inet_timer()`, that you must call at least twice per second if you are using a superloop system. It calls the polling routine for the following protocols:

- IP (for fragment reassembly)
- TCP
- Modem
- PPP
- DHCP Client
- DHCP Server
- DNS Client
- NAT Router
- RIP.

The following code shows a `tk_yield()` function that uses `inet_timer()`.

```
/*
** tk_yield() - this is called whenever the program is looping
** waiting for user (or network) input. It handles the various
** background work needed such as polling alarm conditions and
** de-multiplexing incoming packets.
*/

extern void inet_timer(void);

void
tk_yield()
{
#ifdef IN_MENUS
   kbdio();          /* check for user input for menus */
#endif

   packet_check();   /* check for newly received packets */

   inet_timer();     /* poll all the protocols */

   /* give cycles to optional features */
#ifdef PING_APP
   ping_check();     /* see if ping reply rolled in */
#endif
#ifdef SMTP_ALERTS
   smtpalert_task(); /* email alerter... */
#endif
#ifdef UDPSTEST
   udp_echo_poll();  /* UDP echo client/server */
```



```
#endif
#ifdef TCP_ECHOTEST
   tcp_echo_poll();  /* TCP echo client/server */
#endif
}
```


8.7 testmenu.c

This file adds three commands to the diagnostic menus system:

`fping` Flood ping.

`farp` Flood ARP.

`fcount` Set flood count.

These routines allow you to send a large number of ICMP ECHO requests or ARP requests to a target host. The default is for these routines to send 100 requests in quick succession to the current default host. You can change the number of requests using the `fcount` command, and you can also change the default host using the `host` command.

Note

Flood ARP must only be used on isolated test networks, as it may disrupt network access for any or all other hosts on the network.

8.8 userpass.c

The `userpass.c` file contains code to implement a simple user/password database. It implements the following functions:

- *add_user()*
- *check_permit()* on page 8-25.

8.8.1 add_user()

This function is used to add a username and password to the database.

Syntax

```
int add_user(char *username, char *password, void *permissions)
```

where:

`username` Is the name of the user to add.

`password` Is the user password.

`permissions` Is unused (see *Usage* below).

Return value

Returns one of the following:

`TRUE` If the user/password combination was accepted.

`FALSE` If the user/password combination was not accepted.

Usage

The `username` and `password` combination specified is added to the table of valid users. The `permissions` parameter is ignored in the example applications and in this library implementation. However, it could be used for checking that the user has the required access permissions for the operation requested.

The number of users is limited by the value `NUM_NETUSERS`, defined in `userpass.h`. The maximum user name and password length are limited by the value `MAX_USERLENGTH`, defined in `userpass.h`.

8.8.2 check_permit()

This function is used to authenticate users.

Syntax

```
int check_permit(char *username, char *password, int appcode, void *permissions)
```

where:

`username` Is the name of the user to check.

`password` Is the password that the user has entered.

`appcode` Is the application asking for authentication (see *Usage* below).

`permissions`
is unused (see *Usage* below).

Return value

Returns one of the following:

`TRUE` If the user has been successfully authenticated.

`FALSE` If the user was not authenticated.

Usage

The `check_permit()` function is called by an application when it wishes to check if a particular user is authorized to use that application. The `username` and `password` values are as entered by the end user, and the `appcode` is selected from the list in `userpass.h`. Like `add_user()`, the permissions parameter is unused in the example applications and in this library implementation, but can be used for checking that the user has the required access permissions for a specific operation, such as writing a system file.

The example implementation of `check_permit()` simply performs a string comparison of the password and username, and if the two match, it is regarded as a positive authentication. Your implementation must implement a proper password authenticating mechanism in order to be secure.

Chapter 9
Internal Functions

This chapter contains a list of internal routines that may be useful to programmers writing customized applications with the stack. These functions are a subset of the routines in the libraries that could be of interest to a TCP/IP application or network interface writer. It contains the following sections:

9.1 ARP routines

The ARP routines are located in `et_arp.c` and are as follows:

- *etainit()*
- *make_arp_entry()*
- *arprcv()* on page 9-3.

9.1.1 etainit()

The `etainit()` routine must be called once at initialization time to initialize the ARP layer. It registers the ARP types with the hardware drivers and sets up an ARP timer.

Syntax

```
int     etainit (void);
```

Return values

`0`	If successful.
`1`	If the ARP type could not be registered with the network driver.

9.1.2 make_arp_entry()

The `make_arp_entry()` routine finds the first unused (or the oldest) ARP table entry and makes a new entry to prepare it for an ARP reply.

Syntax

```
struct arptabent *make_arp_entry(ip_addr dest_ip, NET net);
```

where:

`dest_ip`	Is the IP address to make the entry for.
`net`	Is the associated network interface.

Usage

If the IP address already has an ARP entry, the entry is returned with only the time stamp modified. The MAC address of the created entry is not resolved, but is left as zeros. The eventual ARP reply fills in the MAC address.

Return value

`pointer`	a pointer to the selected ARP table entry.

9.1.3 arprcv()

The `arprcv()` routine is the upcall for received ARP packets. It is called by the interface layer.

Syntax

```
int arprcv(PACKET pkt);
```

where:

`pkt` Is the PACKET containing the incoming ARP packet.

Return values

`0` If the received ARP packet was processed successfully.

`ENP_ code` If not successful (see *ENP_ error codes* on page A-2).

9.2 IP routines

The IP routines are located in various files in the `inet` directory and are as follows:

9.2.1 ip_write()

The `ip_write()` routine fills in the Internet header in the packet and sends the packet through the appropriate net interface.

Syntax

```
int ip_write(u_char prot, PACKET p);
```

where:

`prot` Indicates which protocol the packet is carrying (TCP, UDP, ICMP).

`p` Is the packet to send.

Usage

This routine uses routing. You call it with the `p->nb_plen` and `p->nb_prot` fields set to the start of the upper (UDP) layer, and with the `p->fhost` field set to the target IP address.

Return values

Return one of the following:

`0` If the transmission was successful.

`ENP_SEND_PENDING`
If it is waiting for ARP.

`ENP_ code` If an error is detected (see *ENP_ error codes* on page A-2).

Location

The `ip_write()` routine is found in the following file:

`inet\ip.c`

9.2.2 ip2mac()

The `ip2mac()` routine takes as input an outgoing IP packet with no MAC information and tries to resolve an Ethernet address matching the passed IP address.

Syntax

```
int ip2mac(PACKET pkt, ip_addr dest_ip);
```

where:

`pkt` Is the packet itself, without the MAC address

`dest_ip` Is the IP address of the host or gateway.

Usage

If the MAC address is not already cached, an ARP request is broadcast for the missing IP address. The packet is then attached to the *pending* pointer. The packet is sent when the ARP reply comes in, or it is freed if the request times out.

Return values

Returns one of the following values:

`SUCCESS (0)`
: If the packet was sent.

`ENP_SEND_PENDING`
: If awaiting the ARP reply.

`SEND_FAILED`
: If an error was detected.

Location

The `ip2mac()` routine is found in the following file:

`inet/ipnet.c`

9.2.3 ip_mymach()

The `ip_mymach()` routine returns the address of your machine, relative to a given foreign host IP address.

Syntax

```
ip_addr ip_mymach(ip_addr host);
```

where:

`host` Is the IP address of the foreign host.

Usage

On a single-home host, this always returns the IP address of the sole interface. On a router, it returns the address of the interface where packets for the host are routed.

Return value

This routine returns the IP address of the interface used to send packets to `host`.

Location

The `ip_mymach()` routine is found in the following file:

```
inet\ip.c
```


9.2.4 iproute()

The `iproute()` routine performs IP routing on an outgoing IP packet.

Syntax

```
NET iproute(ip_addr host, ip_addr *hop1);
```

where:

`host` Is the IP address of the final destination host.

`*hop1` Is the IP address to use in resolving the MAC address.

Usage

This routine takes the destination Internet address for a packet, and returns the net interface through which to send it.

An IP address is returned (pointed to by the output parameter `hop1`). This is the IP address for resolving the MAC destination address of the packets. If the target host is on the local segment, `hop1` is the same as `host`. Otherwise, it is the IP address of the gateway or router through which `host` is accessible.

Return value

Returns one of the following values:

`pointer` Indicates a net structure which describes the interface of the MAC media to use for sending the packet.

`NULL` If unable to route.

Location

The `iproute()` routine is found in the following file:

```
inet\ip.c
```


9.2.5 add_route()

The `add_route()` routine makes an entry in the route table. If the route already exists, it is updated.

Syntax

```
RTMIB add_route(ip_addr dest, ip_addr mask, ip_addr nexthop, int iface, int
prot);
```

where:

`dest` Is the destination network or host IP address.

`mask` Is either the subnet mask for the destination network, or `0xFFFFFFFF` if the destination is a host.

`nexthop` Is the IP address of the next hop router.

`iface` Is the number of the network interface used to reach the next hop router.

`prot` Indicates the source of the route:

`IPRP_RIP`	Is set through RIP.
`IPRP_HELLO`	Is set through HELLO protocol.
`IPRP_GGP`	Is set through GGP.
`IPRP_EGP`	Is set through EGP.
`IPRP_ICMP`	Is set through ICMP redirect.
`IPRP_NETMGMT`	Is set through SNMP.
`IPRP_LOCAL`	Is set manually.
`IPRP_OTHER`	Is none of the above.

Return value

This routine returns a pointer to the table entry, so the caller can process it further, for example, add metrics.

Location

The `add_route()` routine is found in the following file:

`inet\ip.c`

9.2.6 ip_rcv()

The `ip_rcv()` routine is the IP receive upcall routine. It handles packets received by network ISRs, and so on, verifies their IP headers, and performs the upcall to the upper layer that receives the packet.

Syntax

```
int ip_rcv(PACKET p);
```

where:

`p` Is the received packet.

Usage

You call this routine with `p->nb_prot` and `p->nb_plen` pointing to the start of the IP header, and with the MAC information fields filled in.

Return values

Returns one of the following values:

`0`	If the packet was processed successfully.
`ENP_NOT_MINE`	If the packet was not for this destination.
`ENP_ code`	If the packet was badly formed (see *ENP_ error codes* on page A-2).

Location

The `ip_rcv()` routine is found in the following file:

`inet\ipdemux.c`

9.2.7 parse_ipad()

The `parse_ipad()` routine looks for an IP address in a buffer, and forms an IP address (in network byte order) from it.

Syntax

```
char *parse_ipad(ip_addr *ipout, unsigned *sbits, char *stringin);
```

where:

`*ipout` Is a pointer to the IP address to set.

`*sbits` Is a pointer to a location that is filled in with the number of bits set in the default subnet mask for `ipout`. Its value is 8, 16, or 24.

`*stringin` Is the buffer containing the ASCII to parse.

Return values

Returns one of the following values:

`NULL` If the operations was successful.

`pointer` A string that describes the syntax problem in the input string.

Location

The `parse_ipad()` routine is found in the following file:

`\misclib\parseip.c`

9.2.8 pk_alloc()

The pk_alloc() routine allocates a netbuf structure and associated packet buffer that the caller can use to store data to be transmitted or data that has been received.

This routine is used internally by the stack to pass data between the various protocol layers.

Syntax

```
PACKET pk_alloc(unsigned int len);
```

where:

len — Is the length in bytes of the packet data to be stored in the buffer.

Usage

You must lock the FREEQ_RESID before calling pk_alloc(), and unlock it after pk_alloc() returns.

Return values

Returns one of the following values:

pointer — If the allocation was successful, a pointer to the allocated netbuf structure is returned.

NULL — If allocation was unsuccessful.

Location

The pk_alloc() routine is found in the following file:

inet\pktalloc.c

9.2.9 pk_free()

The `pk_free()` routine returns a previously allocated `netbuf` structure to the pool of such structures that is maintained by the stack.

Syntax

```
void pk_free(PACKET pkt);
```

where:

`pkt` Is a pointer to the `netbuf` structure previously allocated by `pk_alloc()`.

Usage

Include a call to `pk_free()` in your network interface code in order to return a `netbuf` structure and its associated packet buffer to the free pool, after the packet has been transmitted by the network device. For a description of how this is performed, see the description of *pkt_send()* on page 2-22.

You must lock the `FREEQ_RESID` before calling `pk_alloc()`, and unlock it after `pk_alloc()` returns.

Return values

None

Location

The `pk_free()` routine is found in the following file:

`inet\pktalloc.c`

9.3 ICMP routines

The ICMP routines are located in various files in the `inet` directory and are as follows:

- *icmprcv()*
- *icmp_destun()* on page 9-15
- *icmpEcho()* on page 9-16.

9.3.1 icmprcv()

The `icmprcv()` routine is the ICMP received packet upcall handler.

Syntax

```
int icmprcv(PACKET p);
```

where:

`p` Is the received packet.

Usage

Call this routine with `p->nb_prot` and `p->nb_plen` pointing to the start of the ICMP header, and with `p->fhost` filled in.

Returned values

Returns one of the following values:

`0`	If the packet was processed successfully.
`ENP_NOT_MINE`	If the packet was not for this destination.
`ENP_ code`	If an error occurred (see *ENP_ error codes* on page A-2).

Location

The `icmprcv()` routine is found in the following file:

```
inet\icmp.c
```


9.3.2 icmp_destun()

The `icmp_destun()` routine sends an ICMP *destination-unreachable* packet.

Syntax

```
void icmp_destun(ip_addr host, struct ip *ip,
                 unsigned type, NET net);
```

where:

`host` Is the destination IP host.

`*ip` Is the IP header of the packet which triggered the *destination-unreachable* packet.

`type` Is one of the ICMP *destination-unreachable* message types. It must be one of the following defined constants:

- `DSTNET`
- `DSTHOST`
- `DSTPROT`
- `DSTPORT`
- `DSTFRAG`
- `DSTSRC`.

`net` Is the interface that the packet came in on.

Returned values

None.

Location

The `icmp_destun()` routine is found in the following file:

`inet\icmp.c`

9.3.3 icmpEcho()

The `icmpEcho()` routine sends an ICMP echo request.

Syntax

```
int icmpEcho(ip_addr host, char *data, unsigned length, unshort pingseq);
```

where:

`host` Is the host to ping (32-bit, local-endian).

`*data` Is the ping data. This is set to NULL in a *don't care* case.

`length` Is the total desired length of the packet on media.

`pingseq` Is the ping sequence number.

Usage

This routine is callable from applications. It sends a single ping (ICMP echo request) to the specified host. The application must provide an appropriate `pingDemux()` routine if ping replies are to be checked.

Return values

Returns one of the following values:

`0` If the transmission was successful.

`ENP_ code` If an error occurred (see *ENP_ error codes* on page A-2).

Location

The `icmp_destun()` routine is found in the following file:

`inet\ping.c`

Appendix A
Error Codes

This appendix contains a list of both the standard `ENP_` error codes you might encounter while porting (see Chapter 2 *TCP/IP API Functions* and Chapter 5 *Low-overhead UDP Functions*) and the socket error codes (see Chapter 4 *Sockets*). It contains the following sections:

A.1 ENP_ error codes

The error codes listed in Table A-1 are used throughout the stack. Success is zero, definite errors are negative numbers, and indeterminate conditions are positive numbers. These codes are provided in `ipport.h`. You can modify them to wrap around an existing system.

Note

If you define errors with non-negative values, the stack does not work.

These error codes are typically returned by functions that return an integer. See the function specifications (Chapter 2 *TCP/IP API Functions* and Chapter 5 *Low-overhead UDP Functions*) for information on specific functions.

Table A-1 ENP_ error codes

Error type	Error code (defined in `ipport.h`)	Return value	Description
No errors:	`SUCCESS`	0	Success
	`OK`	0	Success
Nonfatal/success:	`ENP_SEND_PENDING` `ARP_WAITING`	1	ARP is holding the packet while awaiting a response from the target host
	`ENP_NOT_MINE`	2	The packet was of no interest (callback reply only)
Programming errors:	`ENP_PARAM`	–10	Bad parameter
	`ENP_LOGIC`	–11	Illogical sequence of events
System errors:	`ENP_NOMEM`	–20	`malloc()` or `calloc()` failed
	`ENP_NOBUFFER`	–21	Ran out of free packets
	`ENP_RESOURCE` `SEND_DROPPED`	–22	Ran out of queueable resources OR full queue
	`ENP_BAD_STATE`	–23	TCP layer error
	`ENP_TIMEOUT`	–24	Operation did not complete in reasonable time
	`ENP_NOFILE`	–25	Expected file was missing.
	`ENP_FILEIO`	–26	File I/O error

Table A-1 ENP_ error codes (continued)

Error type	Error code (defined in `ipport.h`)	Return value	Description
Net errors:	`ENP_SENDERR`	–30	Send to net failed at a lower layer
	`ENP_NOARPREP`	–31	No ARP reply for a given host
	`ENP_BAD_HEADER`	–32	Bad header at the upper layer (for callbacks)
	`ENP_NO_ROUTE`	–33	Cannot find a reasonable next IP hop
	`ENP_NO_IFACE`	–34	Cannot find a reasonable interface
	`ENP_HARDWARE`	–35	Detected a hardware failure

A.2 Socket error codes

Table A-2 lists the sockets errors that may be encountered when implementing ARM sockets. They are a subset of the standard Berkeley errors. See the function specifications in Chapter 4 *Sockets* for information on specific functions.

Table A-2 Socket error codes

Error code	Return value	Description
ENOBUFS	1	Insufficient packet buffers available to complete the operation
ETIMEDOUT	2	The operation could not be completed within the time limit
EISCONN	3	A connection is already established, so a new one cannot be established at this time
EOPNOTSUPP	4	The requested operation, protocol, or format is not supported
ECONNABORTED	5	The connection or connection attempt was aborted
EWOULDBLOCK	6	The requested operation would have to block in order to complete and the socket has been marked as nonblocking
ECONNREFUSED	7	The attempted connection has been refused by the remote host
ECONNRESET	8	The connection associated with this socket has been reset
ENOTCONN	9	The requested operation cannot be completed because the socket is not in the connected state
EALREADY	10	The requested operation cannot be performed because a similar operation is already in progress on this socket
EINVAL	11	The requested operation is invalid in the current socket state, or one or more of the arguments for the request is invalid
EMSGSIZE	12	The datagram is too large to be sent
EPIPE	13	Cannot send using this socket because it has been shutdown for writing
EDESTADDRREQ	14	An address must be specified for `t_connect()` to connect to
ESHUTDOWN	15	The operation could not be completed because the socket has been shutdown

Table A-2 Socket error codes (continued)

Error code	Return value	Description
ENOPROTOOPT	16	The option that you have requested or tried to set using `t_setsockopt()` or `t_getsockopt()` has not been recognized
EHAVEOOB	17	There is Out Of Band data waiting on the socket
ENOMEM	18	The socket sub-system could not allocate enough memory to complete the requested operation
EADDRNOTAVAIL	19	The requested address is not available
EADDRINUSE	20	The requested address is already in use
EAFNOSUPPORT	21	The only address/protocol family supported is AF_INET
EINPROGRESS	22	The connect request failed because a previous connect was already in progress
ELOWER	23	There was an error in the IP layer.

Appendix B
Editing ARM Networking .nv Files

This appendix describes the values that you can specify in `.nv` files. It contains the following sections:

B.1 About the .nv files

Some ARM networking products require various types of configuration to occur on the target system at run time. For example:

- TCP/IP needs to know either its own IP address or whether an IP address is to be picked from a DHCP server.
- DHCP server needs to know the IP address pools.

These values can be specified in `.nv` files. These files are plain text files that can be easily read and modified by a human operator. Each data item occupies one line of text. The name of the data item is first on the line. Every data item name ends with a colon character. The text after the colon is the data; usually an IP address, numeric parameter, or text string.

Note

A pattern-match is performed on the parameter names, so the end-user must be informed that these must not be changed.

All sample applications (Appendix C *Sample Applications*) read parameters from a primary `.nv` file and some of these applications also read parameters from secondary `.nv` file(s).

Primary .nv file

The primary file used depends upon the options that have been defined when building a target. Any one of the following files may be used:

`ether.nv`	Used when USE_PPP is undefined, that is, for ethernet-compatible targets.
`direct.nv`	Used when USE_PPP and DIRECT_RAS are defined, that is, where PPP is used over a serial link.
`dialup.nv`	Used when USE_PPP is defined and DIRECT_RAS is undefined, that is, where PPP is used over a modem.

The parameters documented in *Primary .nv file parameters* on page B-3 are all for use in primary `.nv` files.

Secondary .nv file(s)

The secondary `.nv` file(s) can be used by an application, depending on which products are implemented in the application. Secondary `.nv` files and their contents are described in *Secondary .nv file parameters* on page B-6.

B.2 Primary .nv file parameters

This section describes parameters that are used in the primary `.nv` file. These parameters are categorized according to the products that they support. These products are:

- TCP/IP
- DNS Client
- DHCP Server
- PPP
- Modem
- SNMP
- Webserver.

The parameter names are self-explanatory to experienced TCP/IP programmers.

B.2.1 TCP/IP

For TCP/IP, the following parameters are needed for each of the interfaces:

```
   Net interface: 0
IP address: 10.0.0.1
subnet mask: 255.0.0.0
gateway: 0.0.0.0
DHCP Client: NO
```

Note

The `Net interface:` parameter specifies the interface to which the following parameters apply. If the IP address has to be dynamically assigned via a DHCP Server, the DHCP Client must have the value `YES`.

B.2.2 DNS Client

To use the DNS Client, you initialize the following fields. Each parameter contains the IP address of a DNS Server.

```
DNS server: 1 - 204.156.128.1
DNS server: 2 - 204.156.128.10
DNS server: 3 - 204.156.128.20
```

The maximum number of DNS servers is defined by `MAXDNSSERVERS` in `dns.h`.

B.2.3 B.2.3 DHCP Server

To use the DHCP Server, you set the following parameter:


```
Be local DHCP server: YES
```

This activates the DHCP Server. DHCP Server then picks the rest of the initialization values from the file `dhcpsrv.nv` as described in *Secondary .nv file parameters* on page B-6.

The DHCP Server also needs the file `dhcprecs.nv` for data storage, as described in *Secondary .nv file parameters* on page B-6.

B.2.4 B.2.4 PPP

To use PPP, you set the following parameters:

```
PPP Console Logging: YES
PPP File Logging: NO
PPP keepalive: 0
PPP client timeout: 60
```

If VJ Compression is enabled in the source code, set:

```
PPP VJ request: YES
```

If CHAP is enabled in the source code, set:

```
CHAP secret: secret_words
require CHAP: NO
```

If PAP is enabled in the source code, set:

```
require PAP: NO
```

B.2.5 Modem

To use the modem code, set the following values:

```
Phone Number: your_isp
User Name: your_name
Password: your_password
Modem Init: AT&D2&C1
Idle line timeout: 600
login file: login.nv
log server file: server.nv
```


B.2.6 SNMP

If SNMP agent is enabled, the following system parameters are needed:

```
SNMP Get Community: public
SNMP Set Community: public
SNMP sysContact: Somebody
SNMP sysName: ARM_networking_print_server
SNMP sysLocation: Lab
SNMP Trap target1: 10.0.0.85
SNMP Trap Community1: public
```

B.2.7 Webserver

Webserver needs to know the directory where HTML content is located:

```
http root: /
```

B.3 Secondary .nv file parameters

This section describes parameters that are used in secondary `.nv` files. These parameters are categorized according to the .nv file that they are used in. These files are:

`dhcpsrv.nv` This file is described in Porting the *ARM DHCP Server Version 1.6 Programmer's Guide*, supplied separately with the ARM DHCP Server product.

`dhcprecs.nv` This file is described in *Porting the ARM DHCP Server Version 1.6 Programmer's Guide*, supplied separately with the ARM DHCP Server product.

`login_file.nv` The name of this file is specified after `login file:` in the primary `.nv` file. The use of this file is described with the `do_script()` function in the *Porting PPP Version 1.6 Programmer's Guide*. This guide is supplied separately with the ARM PPP product.

Appendix C
Sample Applications

This chapter describes the sample applications provided with the TCP/IP sources. It contains the following sections:

C.1 Requirements

In order to use the sample projects supplied, you need the following:

- ARM Development Suite (ADS), version 1.0.1
- ARM Integrator/AP fitted with an Integrator/CM7TDMI core module
- ARM Ethernet Kit for the ARM Development Board (Intel PRO/100+ Management Adapter)
- ARM MultiICE for downloading to the board.

If you are using the ARM PPP sources, you also need a Hayes-compatible modem and a PPP server to dial into.

C.2 Building projects

A single CodeWarrior project file is supplied for each example application. These project files have an `.mcp` file extension. Project files can be used to build one or more targets.

All project files build a target supporting 16-bit Thumb code for a little-endian system, full debug and Ethernet connectivity. Some project files also provide a second target that uses PPP rather than the Ethernet drivers.

Note

Take care when selecting options in CodeWarrior, as any changes made are saved immediately to the project file. If the project file is closed and reopened, any changes made in the earlier invocation persist. For more information on using CodeWarrior, see the *ADS CodeWarrior IDE Guide*.

C.2.1 Project files

The project files must be opened with CodeWarrior. There are two ways to open the files:

- Select the project file from Windows Explorer, assuming that the appropriate **Open With** link to CodeWarrior is made.
- Run CodeWarrior, and select the Project File from the **File** → **Open** menu option.

When the project is open, a project window is displayed. If there is more than one target, use the drop-down list to select the target required (PPP or Ethernet).

You can then build the executable image by selecting **Make** from the **Project** menu, or pressing Function Key F7.

C.2.2 Project folders

Working folders are used for each project. These are created for each CodeWarrior project when it is loaded in the folder containing the `.mcp` file. The folder has the same name as the project file, with `_Data` appended. Another sub-folder is also created for each target in the project. This holds the executable `.axf` and another folder that contains the object files for the target.

C.2.3 Cleaning up after a build

To clean up completely after a build, delete the folder called `target_Data` from the folder containing the project file.

C.3 Running the examples

Before you can run the program on the Integrator, you must:

1. Start Multi-ICE and configure it using Auto-Configure.
2. Use a cross-over cable to connect Serial Port A of the Integrator/AP to a `COM:` port on your workstation.
3. Use a terminal emulator to monitor the `COM:` port for diagnostic messages.

Serial A is configured to run at 38400 baud, and uses 8 data bits, 1 stop bit, and no parity.

For PPP examples, connect your modem to Serial Port B on the Integrator/AP.

You can download the target into the Integrator using CodeWarrior by selecting **Debug** from the CodeWarrior **Project** menu. The startup commands and sequences for AXD, the debug tool, are defined in the CodeWarrior project, and use command files provided.

The operation of CodeWarrior, MultiIce and AXD are described in the documentation provided with the *ARM Developer Suite* (ADS).

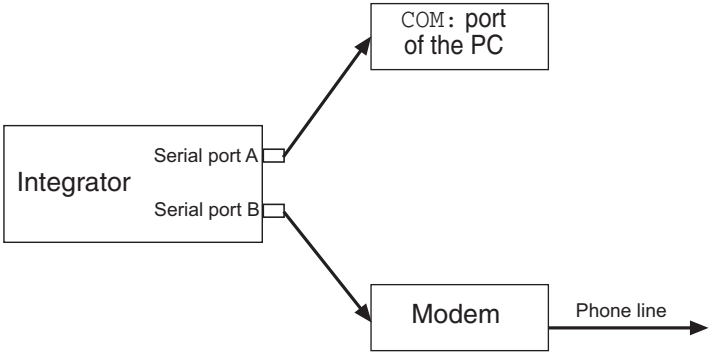

Figure C-1 Running PPP examples

C.4 Descriptions of the examples

The following sections contain detailed descriptions of each sample application. The demonstration projects shipped with the sources are:

chargen A simple server application.

maildemo An implementation of a simple SMTP client that sends an email message.

menus An interactive menus system that is useful for debugging during porting.

C.4.1 chargen

The `chargen` project provides a simple demonstration of how to implement a server. The `chargen` server listens for connections on TCP port 19. When a connection is established, `chargen` continuously sends lines of test data (a swirling printer test pattern, with line numbers).

When the connection is closed, the `chargen` server returns to waiting for connections.

C.4.2 maildemo

The `maildemo` project demonstrates how to write a simple client program. The client:

- connects to the SMTP port (TCP port 25) of a known mail server
- exchanges some messages with the sendmail that is listening on that port
- sends an email message to a preconfigured destination.

To configure the mail server IP address and the `To:` and `From:` addresses of the email message, modify the values defined near the top of `\maildemo\main.c` before you build the project.

C.4.3 menus

The `menus` project is intended as a debugging aid during porting. It presents a menu of options that allow you to exercise different sections of the stack in a controlled manner.

The `menus` project cannot be used with the debug console because `menus` relies on being able to poll the keyboard for data and RDI does not support this operation. To work around this limitation, the UART driver is used as a debug interface by including `ttyio.c` and `uart.c` in the project, and by configuring `ipport.h` to map the print functions onto those implemented as functions in `ttyio.c` and `uart.c`.

Appendix D
The i8255x Ethernet Driver

This appendix describes the i8255x Ethernet Driver for the ARM Network Protocol Suite. It contains the following sections:

D.1 About the i8255x driver

The Intel 10/100 Mbit Ethernet Family is a common, embeddable, PCI Ethernet chip set, that is also used in the Intel PRO/100+ series of PCI Ethernet cards. The range includes the 82557, 82558, and 82559 controllers, that can be coupled with various PHY parts.

The ARM i8255x driver interfaces between any of these controllers and the ARM Network Protocol Suite. It was developed on the Integrator/AP development platform, using several different 8255x cards, version 1.1 of the Intel 8255x documentation, and the ARM Firmware Suite, version 1.1.

It is intended to be easily portable to target hardware.

This section describes particular features of the i8255x driver.

Compatibility The i8255x driver uses none of the additional features of the i82558 and i82559 controllers, and is therefore compatible with the whole range.

PCI bus The PCI Library from the ARM Firmware Suite is used to find all the i8255x devices on the Integrator PCI bus. Multiple devices are allowed, and each device is mapped to a separate interface in the TCP/IP stack.

Note

Early builds of the FPGAs on the Integrator had problems that affected the PCI bus. Ensure that up-to-date versions are used.

Memory structure

Both receive and transmit are performed in *flexible* mode, meaning that the descriptors for the frames are separate from the descriptors for the data buffers. There is one data buffer for each frame.

Contrary to the documentation for the controllers, the end-of-list bits are not set in either the frame descriptors or the buffer descriptors. Rather, the descriptors are linked together as a circular FIFO, and the device never enters a *No Resources* condition. If there are no free descriptors, frames are simply discarded.

Missing features

No power-management features are implemented, TCO functionality is not used, and there is no support for the Intel Adaptive Technology. The checksum offload feature of the 82559 is not used.

Other features

The Intel-suggested workaround for a receiver lock-up problem with some controllers is implemented. This requires a call-back every second from the timer tick handler. This also updates the statistics.

There is a function to put a particular device into promiscuous mode, but this is not currently used.

D.2 Build options

This section describes the build options for the i8255x driver:

- *Statistics*
- *Memory architecture*
- *Other tuneable values* on page D-5.

Note

ARM currently provides the i8255x driver in object form. Consequently, build options are not tuneable. Please refer to `.../readme.txt` to find out about the options used when building the i8255x drivers supplied with this release.

D.2.1 Statistics

You can include code in the build to gather statistics on Ethernet errors from the controller. The code also keeps totals of bytes transmitted, for example. This is controlled by the macro `NET_STATS` in `ipport.h`.

D.2.2 Memory architecture

The driver requires a pool of memory that is accessible by both the CPU and the Ethernet controller. This must be uncached, unbuffered memory.

There are two architecture options available to the driver:

- shared packet
- private buffer.

The pre-processor macro `USE_I8255X_SHARED_PACKETS` selects between them.

If the shared packet architecture is in use (`USE_I8255X_SHARED_PACKETS` is defined in `ipport.h`), all of the large packets (`bigbufsiz`) on the IP stack must reside in an area of memory that can be accessed by both the Ethernet MAC attached to the PCI bus, and the processor. This memory must be uncached and unbuffered. The Ethernet MAC uses DMA to directly read or write the data in the packets buffer area. The advantage of the shared packet architecture is that data need not be copied into the IP stack buffers. The disadvantage is that the packet memory is uncached, and so all accesses to packet structures (for example, for checksum calculation) are external accesses.

If the private buffers architecture is in use (`USE_I8255X_SHARED_PACKETS` not defined), the Ethernet MAC has dedicated memory assigned to it for the transmit and receive buffer space, and data is copied in to or out of the packets on the IP stack using `memcpy()`. The dedicated memory must still be uncached and unbuffered, but the pool of packets on the IP stack can be fully cached.

As shipped, the driver can use either the 512KB of SSRAM on the Integrator/AP, at address `0x28000000`, or the 256KB of unused core module SDRAM at `0x80000000`. Uncomment either `USE_I8255X_SDRAM` or `USE_I8255X_SSRAM` in `i8255x.h`. If you are using a cached processor, ensure that the cacheable and bufferable bits are clear in the appropriate MMU descriptor or MPU register.

D.2.3 Other tuneable values

The other values you can change are:

`MAX_I8255X`

This sets the number of i8255x devices that can be supported by the driver. Each device configured requires a static data structure, so you adjust this value to reflect the actual number of devices in your system.

`I8255X_TX_BUFFERS`

This number governs the number of transmit descriptors and buffers used by the driver. Higher values allow higher performance, at a cost of using more memory.

`I8255X_RX_BUFFERS`

This number governs the number of receive descriptors and buffers used by the driver. If you are using the shared packet architecture, one `bigbufsiz` packet is pre-allocated per receive descriptor, so you must ensure that the system has more than `I8255X_RX_BUFFERS` `bigbufsiz` packets available. Higher values for `I8255X_RX_BUFFERS` allow higher performance and the lower possibility of having to drop incoming packets, but at a cost of using more memory.

D.3 Porting the i8255x driver

This section lists considerations for porting the i8255x driver:

- *Driver memory allocation*
- *µHAL*.

Note

ARM currently provides the i8255x driver in object form. There is no source code provided as a starting point for porting.

D.3.1 Driver memory allocation

Data areas that need to be shared between the Ethernet MAC and the IP stack are allocated using the `i8255x_alloc()` function. As shipped, this function manages the area of memory being used as a heap, using a very simple allocation algorithm that assumes the memory is never freed.

`TO_PCI_ADDRESS` This macro translates from the virtual address of a memory area allocated by `i8255x_alloc()` to the physical address of the memory area, as seen from the PCI bus.

`TO_CPU_ADDRESS` This macro translates the physical address (as seen from the PCI bus) of a memory area to the virtual address that the processor would use to access that memory.

D.3.2 µHAL

If your target system does not support µHAL, you must provide routines that implement equivalent functionality to the following ARM Firmware Suite 1.1 functions:

- `uHALr_PCIHost()`
- `PCIr_ForEveryDevice()`
- `uHALr_EnableInterrupt()`
- `uHALr_DisableInterrupt()`
- `uHALr_FreeInterrupt()`
- `uHALr_RequestInterrupt()`
- `uHALr_PCICfgWrite16()`
- `uHALr_PCICfgRead16()`
- `uHALr_PCICfgWrite32()`
- `uHALr_PCICfgRead32()`
- `uHALr_PCICfgWrite8()`

- `uHALr_PCICfgRead8()`
- `uHALr_PCIIORead32()`.

Please refer to the documentation that accompanies the ARM Firmware Suite for details of the functionality provided by these routines.

Glossary

ADS ARM Developer Suite.

API Application Program Interface.

ARP Address Resolution Protocol.

AXD ARM eXtendable Debugger.

DHCP Dynamic Host Configuration Protocol.

DMA Direct Memory Access.

DNS Domain Name System.

EOT End Of Transmission.

FIFO First-In, First-Out.

FTP File Transfer Protocol.

HTML Hypertext Markup Language.

HTTP Hypertext Transfer Protocol.

ICMP Internet Control Message Protocol.

IP Internet Protocol.

ISR Interrupt Service Routine.

LAN Local Area Network.

MAC Media Access Control.

MIB Management Information Base.

MMU Memory Management Unit.

MPU Memory Protection Unit.

NIC Network Interface Controller.

NVRAM Non-Volatile Random Access Memory.

PPP Point-to-Point Protocol.

RDI Remote Debug Interface.

RFC Request For Comments.

RIP Routing Information Protocol.

RTOS Real-Time Operating System.

SDRAM Synchronous Dynamic Random Access Memory.

SDT Software Development Toolkit.

SLIP Serial Line Internet Protocol.

SMTP Simple Mail Transfer Protocol.

SNMP Simple Network Management Protocol.

SSRAM Synchronous Static Random Access Memory.

TCO Total Cost of Ownership.

TCP Transmission Control Protocol.

TFTP Trivial File Transfer Protocol.

UART Univeral Asynchronous Reveiver/Transmitter.

UDP User Datagram Protocol.

VFS Virtual File System.

VJ Van Jacobson.

Index

The items in this index are listed in alphabetical order, with symbols and numerics appearing at the end. The references given are to page numbers.

A

B

C

D

E

F

G

H

I

L

T

U

W

Directories

Symbols

