Porting PPP

Version 1.6

Programmer's Guide

ARM

ARM DUI 0143B

Porting PPP
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 indepenedent PPP guide (ARM DUI 0143) for software version 1.6
June 2001	B	Second release

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
Porting PPP Programmer's Guide

Preface

This preface introduces the ARM PPP implementation and its documentation. It contains the following sections:

About this book

This guide is provided with the ARM Portable PPP stack sources.

It is assumed that the ARM PPP 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 PPP, 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 PPP stack sources.

Chapter 2 ***PPP Porting***

Read this chapter for a description of PPP, and how to use the ARM PPP code to allow the ARM TCP/IP code to transfer data over serial lines.

Chapter 3 ***PPP API Functions***

Read this chapter for a description of the user-provided functions and other entry points required for porting the ARM PPP stack.

Chapter 4 ***Modem Functions***

Read this chapter to learn how to interface a Hayes-compatible modem to the PPP stack.

Appendix A ***Testing the PPP stack***

Read this appendix to learn how to test the ARM PPP stack against a Windows NT machine.

Typographical conventions

The following typographical conventions are used in this book:

`typewriter` Denotes text that can 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 by both ARM Ltd and third parties that provide additional information that may help with porting ARM PPP.

ARM publications

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

- *ARM Architecture Reference Manual* (ARM DUI 0100)
- *Porting TCP/IP Programmer's Guide* (ARM DUI 0144)
- *ARM ADS Tools Guide* (ARM DUI 0067)
- *ARM ADS Developer Guide* (ARM DUI 0056).

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 1877, Cobb, S., *PPP Internet Protocol Control Protocol Extensions for Name Server Addresses*, December 1995
- RFC 1661, Simpson, W., *The Point-to-Point Protocol (PPP)*, 07/21/1994.

Feedback

ARM Limited welcomes feedback on both ARM PPP and its documentation.

Feedback on ARM PPP

If you have any problems with ARM PPP, 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 stand-alone 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 your comments apply
- a concise explanation of your comments.

General suggestions for additions and improvements are also welcome.

Chapter 1
Introduction

This chapter introduces networking, the ARM porting functions, the requirements for porting ARM PPP, a list of the sample package directories, and an overview of the sample programs provided. It 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.

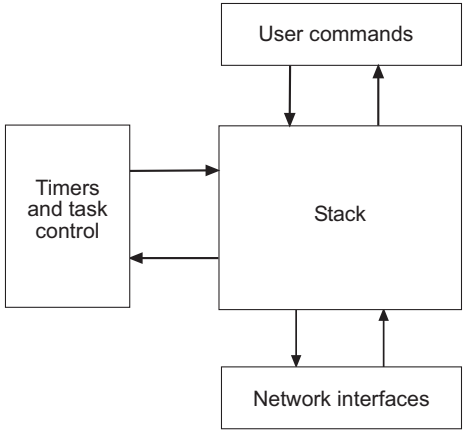

Figure 1-1 Network stack events

In an ideal situation, calls are mapped directly onto the underlying system. For example, when the stack makes an external to send a packet, this has the same syntax as the exported send call of 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.

1.1.1 The ARM portable stack

The ARM PPP 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 PPP stack requires a glue function to write a byte to a particular serial device.

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.

1.2 What is PPP?

PPP is a specification for the transmission of network data over point-to-point links. It:

- converts blocks of network data (packets) into single bytes for transmission over a serial line, such as ISDN or a dial-up modem, and re-assembles the packets on receipt
- checksums the packets
- compresses TCP/IP protocol headers
- verifies the identity of (authenticates) the computer on the other end of the line
- allows packets from multiple protocols to be transferred on a shared line.

PPP does not handle modem dialing. However, ARM provides additional software with PPP that does this for a standard Hayes command-set modem. The ARM PPP stack includes IPCP, to enable the transfer of IP datagrams. It does not include layers for non-IP protocols, such as AppleTalk and DECnet.

Note

In this document, the term PPP, when used without qualification, refers to the ARM PPP code as ported to an embedded system.

1.2.1 Protocols

The ARM package implements all the protocols required for IP transmission and the optional protocols for authentication and TCP/IP header compression.

PPP is actually a family of protocols, all working together to provide the functionality described above. Two members of the family, LCPand IPCP, provide a virtual connection service and handle a set of options, such as the authentication to use and whether to compress packets.

Each connection protocol moves these connections between states defined by a *Finite State Machine* (FSM) specification. Other members of the PPP family provide services to the connection protocols, such as security (CHAP, UPAP).

Figure 1-2 on page 1-5 shows how PPP fits between the IP protocol family and the hardware link. In this case, the line hardware is a modem.

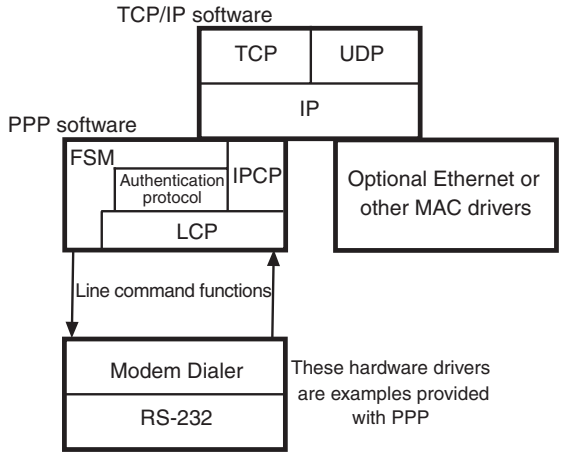

Figure 1-2 The IP protocol and PPP

There are three layers:

LCP Link Control Protocol.

This is the carrier on which all the other protocols are layered. This layer is responsible for establishing the initial link between the two ends, then prompting the upper layers to start their option negotiation. It is also responsible for identifying the protocol of each incoming packet and passing those packets to the appropriate upper layers.

IPCP IP Control Protocol.

This handles IP-related options and packets. IPCP options include assigning IP addresses and negotiating the use of TCP/IP header compression. All IP packets sent and received on the PPP connection are encapsulated in this protocol.

FSM Finite State Machine.

This is not an actual protocol, but contains the definitions for the series of events that a PPP connection protocol moves through, from the initiation of the link to termination.

Both LCP and IPCP use the same FSM code.

For more detail about the individual layers, see RFC 1661, the PPP specification.

Authentication protocols

You can configure any or all of the following authentication protocols, and each end of the link can negotiate which of them will be used:

CHAP Challenge Handshake Authentication Protocol.

CHAP is the primary mechanism for a PPP node to guarantee the identity of the host on the other end of the line. Authentication is initiated by sending a CHAP message (the challenge) through LCP from one PPP host to the other. The CHAP challenge contains an encrypted string, generated by the industry standard MD5 digest algorithm, based on an ASCII string (called a secret) that is known to both hosts.

The challenged host must return the correct CHAP reply. If it does not, the challenger terminates the connection. Generally, either host can send the CHAP challenge at any time after the LCP connection is established.

MS-CHAP The Microsoft implementation of CHAP (see above).

The encrypted string found in the CHAP challenge is generated by a Microsoft proprietary algorithm, rather than by the MD5 digest algorithm. The proprietary algorithm is based on a unicode string that is known to both hosts.

MS-CHAP is used by Microsoft *Remote Access Service* (RAS).

UPAP User/Password Authentication Protocol.

UPAP is similar to CHAP, except that a user name and password are used to generate the authentication packets. This is useful when multiple users with different levels of access might be dialing into a PPP interface.

1.3 ARM PPP requirements

The ARM PPP software requires the following support from the host system:

- *Line management functions*
- *Static memory* on page 1-8
- *Dynamic memory* on page 1-9
- *Periodic clock tick* on page 1-10.

1.3.1 Line management functions

PPP must send and receive characters on the line hardware of the target system. It might also have to initiate a connection (for example, dial the phone number) or disconnect. You must provide a set of low-level functions to do this. If more than one type of line is to be used, such as ISDN and Dialup, a set of functions must be provided for each line type.

PPP defines a structure that contains a set of pointers to these functions. You must ensure that all these pointers are set to appropriate functions at system initialization time, even if the function does nothing other than return. Providing these functions is generally the bulk of the work required to implement PPP on a new target system.

The PPP code comes with two sets of line management functions:

- A *Universal Asynchronous Receiver/Transmitter* (UART) serial line with modem dialer. The line driver calls are described in detail in *Serial line drivers* on page 3-10.
- A loopback driver. This is for testing purposes only and is not expected to be the primary line driver of a real product.

If your target hardware is an embedded system and you intend to use an 8250/16450/16550 (or similar) UART and a Hayes-compatible modem, you can use the (UART) line drivers exactly as provided.

1.3.2 Static memory

As with all embedded system code, the PPP code takes up some code and data space. On embedded systems, the code is usually stored in ROM and can be moved to RAM at boot time. The exact amount of code space required varies depending on:

- the PPP optional features you enable
- your processor
- your compiler.

Table 1-1 on page 1-9 and Table 1-2 on page 1-9 provide sample sizes of the major modules in the sample program compilation. All figures exclude C runtime libraries, board support, and application-specific code.These statistics were obtained under the following conditions:

- compiled with space optimization enabled
- APCS 3 32-bit, no software stack check, no frame pointer
- Linker configured to remove unused sections
- compiled with the `-zo` option to generate one ELF section for each function if the source code
- compiled without debug code.

Note

Because the code is subject to continuous development, these values might change with subsequent releases.

Total static memory for a configuration will be the amounts shown in the following tables plus about 3kB of data space multiplied by the maximum number of connections (`_NPPP`).

Table 1-1 PPP code size (in bytes) with all options enabled

	Thumb Code and read-only data	Read-write data	Zero-init data	Thumb ROM	RAM
PPP	11292	136	104	11428	240
IPCP	5216	60	196	5276	256
LCP	5232	64	144	5296	208
CHAP	6508	64	164	6572	228
MS-CHAP	5764	184	0	5948	184
VJ compression	2396	0	0	2396	0
(U)PAP	1580	0	48	1580	48
Modem	3864	292	292	4156	584
Totals	41852	800	948	42652	1748

Table 1-2 PPP code size (in bytes) with all options except NB_CONNECT disabled

	Thumb code and read-only data	Read-write data	Zero-init data	Thumb ROM	RAM
PPP	9836	104	116	9940	220
IPCP	5200	60	196	5260	256
LCP	5084	64	144	5148	208
Totals	20120	228	456	20348	684

1.3.3 Dynamic memory

PPP has no real dynamic memory requirements. However, because of the way some compilers allocate memory, some of the uninitialized static data areas are allocated at initialization time rather than statically. PPP allocates these areas by calling functions that have the same syntax as a standard C library `malloc()` call. These calls differ from `malloc()` in two ways:

- they expect the returned buffer to be initialized to zeros (like `calloc()`)
- each macro is used for one kind of buffer or structure only, so a reasonable expected maximum size can be defined at compile time.

If your C compiler and development environment support `calloc()`, you can map the allocation macros directly to `calloc()` using the default macro definitions in the sample source code.

If your system does not support `calloc()` or if you do not want to use it for performance reasons, you can reserve arrays of static buffers of the sizes required and return pointers to them from the allocation macros. The exact sizes required vary with the environment (for example, CPU type and compiler packing options), so you must use `sizeof()` operators in your static declaration statements.

The number of buffers of each type varies with the number of lines (units) you can open at once.

1.3.4 Periodic clock tick

The PPP code includes a function that must be called by the system once per second. This function drives retransmissions and timeouts.

In addition, the PPP code expects the system to maintain a 32-bit clock tick counter variable, `cticks`, that increments `TPS` times a second. The macro `TPS` must be defined in your `ipport.h` file or one of its nested includes (see the *Porting TCP/IP Programmer's Guide* for details).

1.4 Sample package directory and programs

The ARM PPP sources are distributed in a file called `ppp.zip`. This unpacks to the following directories:

install_directory`\crypt`

Functions used by CHAP authentication.

install_directory`\modem`

Functions for controlling a Hayes-compatible modem.

install_directory`\ppp`

PPP implementation.

1.4.1 Sample programs

Most of the sample programs provided with the ARM TCP stack can make PPP connections using a standard Hayes modem. For example, you can dial into an Internet Service Provider with the Menus sample program and issue commands to ping remote hosts. If you configure the IP stack to support both Ethernet and PPP, you can use it as a dial-up router. See the *Porting TCP/IP Programmer's Guide* for more information.

The sample code compiles with the *ARM Developer Suite* (ADS). Unless you are familiar with PPP 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.

Chapter 2
PPP Porting

This chapter describes PPP and how to use the ARM PPP code to allow the ARM TCP/IP stack to transfer data over a serial line. It contains the following sections:

2.1 Overview of the porting procedure

To create a working version of the PPP stack on your target system:

1. Copy the PPP source files into your development environment.
2. Modify the `ppp_port.c` and `ppp_port.h` files (see *Source files* on page 2-4).
3. Compile the code (see *Compiling PPP* on page 2-5).
4. Add the hooks to connect PPP to your system (see *Entry points and support calls* on page 2-8).
5. Test a PPP image (see *Testing PPP* on page 2-9).

2.2 Porting PPP

This section outlines the steps you must follow to port the ARM PPP code into the ARM IP stack. Initially, define `USE_PPP` in `ipport.h` and include the PPP sources in your `makefile` or project file (`.mcp`).

You must set up several port-specific defines, functions, and static variables before PPP can operate. These are:

`#define _NPPP 3`

The maximum number of simultaneous PPP connections allowed (for example, 1 per modem).

`void ConPrintf(char *, …);`

A user-provided `printf`-like function for debugging. In the sample program, you can set this up to send its output to the console, a log file, or both. Logging to a file during development is highly recommended if you are writing a new serial driver.

`int ppp_port_init(int unit);`

The hook for per-port initialization. You must provide code in this function to initialize the `nets[]` and `ppp_lines[]` entries.

2.2.1 Source files

As provided, the PPP source code is several C source files and include files. These are called the *portable* or *port-independent* source files. You do not need to modify these for a normal PPP port.

The PPP code is organized into files that are named for the layer or module implemented. For example, `lcp.c` implements the LCP functionality. All connection-oriented modules have a number of functions that implement the FSM. These are table driven and, as such, are compiled into `fsm_callbacks` structures as defined in `fsm.h`. Typically, you do not have to modify these functions.

Two additional files, one C source and one include file, are provided as part of the sample package. These files implement the port-dependent functions. You must duplicate the functionality of these files in the target system as part of the porting process.

The sample port files total approximately 16KB of commented source. They are:

- `ppp_port.c`
- `ppp_port.h`.

If you have licensed the ARM TCP/IP stack, it is recommended that you compile and run the ARM TCP/IP sample programs before you begin your porting activity. This gives you some hands-on experience with PPP and you will have the opportunity to step through the PPP code under the source-level debugger. Also, if your port is unsuccessful, you will have a working reference platform to aid in debugging.

2.2.2 Compiling PPP

The first step in the porting process is to compile the portable portions of the PPP code in your development system. You must set up a `makefile` or project file with the appropriate compile options, library invocations, and linking command.

PPP include file

To compile the code, you must provide your own version of the `ppp_port.h` file to define the data types shown in Example 2-1.

Example 2-1

```
typedef  unsigned char u_char;        /* 8-bit unsigned */
typedef  unsigned short u_short;      /* 16-bit unsigned */
typedef  unsigned short unshort;      /* duplicate */
typedef  unsigned long u_long;        /* 32-bit unsigned */
typedef  int bool;                    /* another common */
                                      /* type extension */
#ifndef  TRUE
#define  TRUE -1
#endif

#ifndef  FALSE
#define  FALSE  0
#endif

#ifndef  NULL
#define  NULL ((void*)0)
#endif

typedef unsigned long ip_addr;       /* 32-bit IP v4 address */
```

For most compilers, you can use these defines exactly as they appear in the sample package `nptypes.h` file, which is in the `..\inet` directory of the ARM TCP/IP sources.

Setting PPP options

Early in the porting process, you must decide which of the optional features of PPP you want to use, and set the defines for them. These defines are generally set in your `ppp_port.h` file. The options they include can be useful, but they can nearly double the size of the PPP code. If your systems have limited memory, you may want to omit these

options. Most ports can simply define VJC and CHAP_SUPPORT, so you can go to *Entry points and support calls* on page 2-8. However, the compile-time options are documented here for completeness.

The following C code excerpt shows all options enabled:

```
#define PPP_VJC             1  /* VJ header compression */
#define CHAP_SUPPORT    1  /* CHAP authentication */
#define PAP_SUPPORT     1  /* password authentication */
#define LOCAL_RAND      1  /* use random number generator */
                           /* in magic.c */
#define LB_XOVER        1  /* cross 2 loopback lines for test */
#define MSCHAP_SUPPORT  1  /* enable Microsoft CHAP */
                           /* authentication */
#define PPP_DNS         1  /* enable RFC1877 operation */
```

Each of these compile switches is described below:

PPP_VJC
: Enables the use of *Van Jacobson Compression* (VJC) to compress TCP/IP headers. VJC is a simple compression algorithm for TCP/IP headers. It is based on the principle that most of the information in the 40-byte TCP and IP headers does not vary on a PPP link from frame to frame. The 40-byte header is replaced with a much smaller header containing only the variable information.

  Because many TCP/IP packets contain only the headers, this can reduce the byte traffic on a PPP link by over 50% before any data compression is applied to the data portion of the packet. The drawback to VJC is that the code is one of the larger modules of PPP. See Table 1-1 on page 1-9.

  Disabling VJC does not prevent your system from operating with any other PPP. It causes both systems to disable the feature and slows performance.

CHAP_SUPPORT
: Includes the code for CHAP and MD5. CHAP must be configured with a secret by the end user and negotiated when LCP connects. If this feature is disabled, you do not have to provide the get_secret() function.

MSCHAP_SUPPORT
: Includes the code for MS-CHAP, using DES and MD4. If this feature is enabled, you must also enable CHAP_SUPPORT.

PAP_SUPPORT
: Includes the code for UPAP. UPAP must be configured by the end user and negotiated when LCP connects.

LOCAL_RAND	This includes code to provide a pseudo-random number generator that is used as part of the compression code. On most target systems, the standard C library calls `rand()` and `srand()` are supported. This option must be enabled to provide these calls on systems that do not already have them.
LB_XOVER	This option applies only if the PPP line loopback driver is used. It configures the loopback driver code to crossover two logical PPP units to each other. Bytes sent on either unit are received on the other crossed-over unit. This provides a testing environment for emulating PPP client/server conditions. This option is normally used only for development and testing and is not needed in the final product. You must define the two unit numbers to be connected. See the loopback example project in the *Porting TCP/IP Programmer's Guide*.
PPP_DNS	Includes code to support the behavior described in RFD1877 (the exchange of DNS addresses as part of the IPCP negotiation.

2.2.3 Entry points and support calls

When you have compiled the PPP code, you must add the hooks to connect PPP to your system and link your line functions to the PPP code. This is generally done in `ppp_port_init()` in the file `ppp_port.c`. *PPP entry points* on page 3-18 provides detailed information about all PPP calls you must be aware of.

There are three classes of functions you must implement:

- Support functions that PPP needs from the host system, for example, time tick, reading NV parameters, and memory allocation.
- Line drivers, functions for sending and receiving bytes, and connection management. These are detailed in *Serial line drivers* on page 3-10.
- IP support functions for sending and receiving IP packets. This is implemented using the NET structure in the ARM TCP/IP stack, so that the IP layer does not treat PPP any differently than it would any other media, such as Ethernet or SLIP.

You must review the list of function calls in *PPP entry points* on page 3-18. At a minimum you need to ensure that:

- `prep_ppp()` and `ppp_timeisup()` are called as appropriate
- the allocation functions are properly mapped
- a line driver is available.

For most ports, the line driver is the majority of the work.

To bind PPP to the ARM TCP/IP stack, you must call `prep_ppp()` from `prep_ifaces()` in your `ipport.c` file. See `prep_ifaces()` in the *Porting TCP/IP Programmer's Guide*. Other IP stacks need a glue layer that maps their initialization, send and close calls to PPP.

2.3 Testing PPP

When you have loaded a PPP image into your target system, you need to test (and possibly debug) it. It is recommended that you perform the following sequence of tests:

- *Loopback*
- *Client connection*
- *Server connection* on page 2-10
- *Abrupt disconnect* on page 2-10
- *Multilink test* on page 2-10.

2.3.1 Loopback

A recommended first test is to set up a PPP loopback driver in crossover mode (see `LB_XOVER` compile option in *Setting PPP options* on page 2-5) and ping it. The recommended IP addresses of your loopback driver interfaces are 127.0.0.1 and 127.0.0.2. The loopback driver must establish an LCP connection between the two crossover units, acting as a client on the unit that sends the ping and a server on the crossover unit. The ping packet exits one unit and enters the other. This is reflected in the packet and byte counters at the IP interface.

In the event the ping does not happen smoothly, the best approach is to trace the execution with the source level debugger. Because all the ping events take place in a single system, you can debug this basic functionality without the need to monitor two separate systems.

2.3.2 Client connection

The next test is to try a client connection through a real line driver to a dial-up server. When the PPP stack has some data to send (perhaps a ping packet), the PPP stack calls `ln_connect()`. When `ln_connect()` returns successfully, the LCP layer sends a series of LCP negotiation packets using `ln_putc()`. LCP will not go to the connected state until it receives the correct LCP responses from the PPP server to which it is connected.

If you have already pinged in loopback, most of the debugging here will probably be in your line driver. For debugging the initial connect call and the first send and receive, a source level debugger is probably the best tool. At the point where LCP packets are being exchanged, you must turn on the logging feature (see *ConPrintf()* on page 3-5) to get a higher level look at what is happening during LCP negotiation.

Debugging LCP when connected with a remote machine is probably the most complex part of a PPP port. It is recommended that you have a copy of the PPP RFC specifications at hand (preferably hardcopy) and you must examine logged LCP option negotiation packets in detail. If logging is enabled, these packets are written to a text file in the current working directory.

For initial testing, turn off CHAP, VJC, and DHCP to simplify the negotiations. Assuming the basic byte transfer works, most LCP problems are because one side of the connection is insisting on an option setting that the other side does not support. When you establish an LCP connection, turn these options on one at a time and retest. You might find you want to add code for option reporting, for example, printing console messages like:

```
Other side insists we use CHAP

Other side refuses to use CHAP
```

2.3.3 Server connection

The next recommended test is that you set your line hardware to auto-answer mode and let another PPP machine call you. Debugging this is similar to client connection debugging.

2.3.4 Abrupt disconnect

You must ensure that a broken connection does not permanently disable your PPP layer. This is usually not a problem when PPP initiates the disconnect by way of a TERMREQ LCP packet, but an unexpected line failure must be sure to call PPP by way of the `lcp_lowerdown()` call (see *lcp_lowerdown()* on page 3-18).

2.3.5 Multilink test

As a final test, test the code with multiple simultaneous links.

Chapter 3
PPP API Functions

This chapter describes the functions you must provide, and other entry points required, for porting the ARM PPP stack. Refer to the code provided with the software for examples.

This chapter contains the following sections:

3.1 Overview of user-provided PPP functions

You must provide the functions described in this chapter as part of porting the ARM PPP stack. In the sample package, these functions are mapped in one of the following ways:

- directly to system calls by way of macros in `ppp_port.h`
- implemented in `ppp_port.c`
- implemented in files in the `\armthumb` and `\integrator` directories.

Many of these implementations map directly onto the system to which you are porting. Others require extensive modification or complete rewrites.

Refer to Chapter 2 *PPP Porting* for the complete PPP porting procedure.

3.2 User-provided PPP functions

This section describes the PPP functions that you must implement:

- *_ALLOC() functions*
- *ConPrintf()* on page 3-5
- *_FREE() functions* on page 3-6
- *get_secret()* on page 3-7
- *ppp_port_init()* on page 3-8.

3.2.1 _ALLOC() functions

These functions allocate a block of memory.

Syntax

```
struct ppp_softc *PPPS_ALLOC(size_t size)

u_char *PPPB_ALLOC(size_t size)

struct timerq *PPPT_ALLOC(size_t size)
```

where:

size Is the number of bytes to be allocated.

Return value

Returns pointers to available memory.

Usage

Each call allocates a single type of buffer of a known maximum size. All have a corresponding `_FREE` function. If your system has a `calloc()` call, these buffers can all be mapped to `calloc()` as shown in the sample `ppp_port.h` file. If you are not using `calloc()`, the buffers returned must have their contents set to all zeros with `memset()` or a similar function.

If your system has no `calloc()` and `free()` functions, you can take buffers from a static partition table as described in *Dynamic memory* on page 1-9.

`PPPS_ALLOC()` and `PPPT_ALLOC()` are only ever called to request structures. The `PPPB_ALLOC()` function is called to request a maximum buffer size given by the expression:

```
length = HDROFF + PPP_MRU + PPP_HDRLEN + PPP_FCS_LEN + 20;
```

where HDROFF is defined in ifppp.c, and PPP_MRU, PPP_HDRLEN, and PPP_FCS_LEN are defined in ppp.h. This can be quite large (over 8KB), but only one block per PPP network interface is allocated at any one time. Changing the definition of PPP_MRU in ppp_port.h can reduce the size, but a size below 1500 bytes is not recommended for interoperability reasons.

In practice, it is rare for a PPPB_ALLOC() request to be larger than 1600 bytes, so in partition table systems with memory limitations, it is usually acceptable to reserve one 1600-byte buffer per PPP network interface.

3.2.2 ConPrintf()

All PPP ports must provide this function to record debug messages to a log. The log can be a file, or UDP log server or other device, such as a serial console. The log can assist you and your end users during complex installations.

Syntax

```
void ConPrintf(const char *format, …)
```

where:

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

… Is an argument list, as described by *format*.

Return value

None.

Usage

In the sample package, this function uses a user-selectable option to print the messages to the system console, to a disk file, or to both. Given the complexities of installing PPP at end user sites, you might find it useful to leave this troubleshooting aid enabled in the final product.

3.2.3 _FREE() functions

These functions free a memory block that was previously allocated. They correspond to the three allocation functions, `PPPS_ALLOC()`, `PPPB_ALLOC()`, and `PPPT_ALLOC()`.

Syntax

```
void PPPS_FREE(struct ppp_soft *)

void PPPB_FREE(u_char *)

void PPPT_FREE(struct timerq *)
```

Return value

None.

Usage

If your system maps the allocation functions to `calloc()`, it must map these functions to `free()`. If you use a partition system, you must mark the returned buffers as free and maintain any data structures required by your algorithm.

3.2.4 get_secret()

This function must be provided for systems supporting CHAP. It gets the CHAP secret stored in NVRAM and makes it available to the PPP CHAP internals.

Syntax

```
int get_secret(int unit, char *resp_name, char *rhostname,      char
*out_buffer, int *out_buflen, int flags)
```

where:

unit Is the PPP unit number. Unit numbers start from zero.

resp_name Points to the name sent with the response.

rhostname Points to the name of the remote host receiving the response.

out_buffer Is the location into which the secret is copied.

out_buflen Points to the location where the number of valid characters in the secret is to be stored by `get_secret()`.

flags Is 0 if you are sending the response to the remote host. It is 1 if you are verifying the response of the remote host to your challenge.

Return value

Returns one of the following:

TRUE If successful.

FALSE If there were problems extracting or copying secret.

Usage

If your target system can initiate or accept connections from multiple remote hosts, you can use *resp_name*, *rhostname*, and *flags* to select the appropriate secret for each connection.The function `get_secret()` copies the secret to `out_buffer` and sets the **`int`** pointed to by `out_buflen` to the number of valid characters. The buffer supplied by the calling function must have room for MAXSECRETLEN characters. The `get_secret()` function must not attempt to write more than this number of characters into the buffer.

3.2.5 ppp_port_init()

This function initializes the interface for each PPP unit. It is called once for each unit from `prep_ppp()`.

Syntax

```
int ppp_port_init(int unit)
```

where:

unit is the PPP unit number to be initialized. Unit numbers start from zero.

Return value

Returns one of the following:

0 If successful.

nonzero error code

If not successful.

Usage

Typically, this function installs the pointers to the line functions in the `ppp_lines[ ]` array, although this can also be done at compile time. In either case, the line functions must be set up when this function returns.

It can also set the defaults for the `ppp_softc` structure of the unit. These include:

`default_ip` This is the default IP address. IPCP can optionally set your IP address for you and it can then be overwritten by DHCP.

If you are not getting the IP address by way of IPCP and are not using DHCP, the default IP address is the operational IP address of your IP stack on this interface.

Because neither IPCP assignment nor DHCP service is universally available, it is advisable to request that the end user assign an IP address (stored in nonvolatile storage) as a fallback. This can be zeros (0.0.0.0) if a DHCP assignment is required at the end user site.

`require_chap` This specifies whether the user configuration requires CHAP security.

If PPP_DNS is defined, these can also include the following, as IPCP can optionally get one or two domain nameserver addresses from the PPP peer for local use. It can optionally pass one or two domain nameserver addresses to the PPP peer for its use:

`dns_pri` Is the default value for the primary nameserver address set in `neg_dns_pri`. This can be zero (0.0.0.0) if there is no default value.

`dns_sec` Is the default value for the secondary nameserver address set in `neg_dns_sec`. This can be zero (0.0.0.0) if there is no default value.

`neg_dns_pri` Is a flag that, if set, indicates that IPCP must try to get the primary nameserver address from the peer.

`neg_dns_sec` Is a flag that, if set, indicates that IPCP must try to get the secondary nameserver address from the peer.

`peer_dns_pri` Is the primary nameserver address to give to the peer.

`peer_dns_sec` Is the secondary nameserver address to give to the peer.

`neg_peer_dns_pri` Is a flag that, if set, indicates that IPCP must try to give a primary nameserver address to the peer.

`neg_peer_dns_sec` Is a flag that, if set, indicates that IPCP must try to give a secondary nameserver address to the peer.

You do not generally have to change the other structure members manually.

3.3 Serial line drivers

The PPP code defines a structure for each serial line it is to use (internally referred to as units):

```
struct com_line         /* structure to direct PPP requests*/
{
    int (*ln_connect)(int unit, struct com_line* lineptr);
                                    /* bring check line up */
    int (*ln_hangup)(int unit);     /* disconnect the line */
    int (*ln_putc)(int unit, int byte);
                                    /* send a byte */
    int (*ln_write)(int unit, char *block, int length);
                                    /* send a buffer */
    long(*ln_speed)(int unit);      /* query line's speed */
    int (*ln_state)(int unit);      /* query line's state */
    int (*ln_getc)(int unit, int byte);
                                    /* receive single char */
    int media_type;                 /* SLIP or PPP */
}
```

An array of these structures (`ppp_lines[_NPPP]`) is statically defined. When the PPP code accesses one of the line functions of the unit, it calls the functions in the table.

You must provide the functions defined in this structure and set pointers to them in `ppp_port_init()`. PPP assumes these functions might block. For example, `ln_connect()` usually dials a phone. Generally, the connect call is the only one that blocks for more than a fraction of a second. PPP does not re-enter the functions or assume any timeout.

If multiple units have the same type of hardware, you can use the same functions in all the `ppp_lines[ ]` entries. However, the line functions must be coded to use the *`unit`* parameter to access the correct hardware device.

The line driver functions are defined in the following sections:

- *ln_connect()* on page 3-11
- *ln_getc()* on page 3-12
- *ln_hangup()* on page 3-13
- *ln_putc()* on page 3-14
- *ln_speed()* on page 3-15
- *ln_state()* on page 3-16
- *ln_write()* on page 3-17.

3.3.1 ln_connect()

This call checks to see whether the line is connected and initiates a connection if the line is unconnected. It typically blocks while the connection is established. This might take a minute or more while a modem line driver dials, awaits an answer, and trains, for example.

When a value of 0 is returned, the PPP code assumes the line is ready to send or receive characters.

Syntax

```
int (*ln_connect)(int unit, struct com_line* lineptr)
```

where:

unit Is the PPP unit number.

lineptr Is a pointer to the com_line structure associated with this *unit*.

Return value

Returns one of the following:

0 Connected and working.

1 Not connected. This is a temporary problem (for example, line busy).

2 Broken, noncorrectable hardware error detected.

3.3.2 ln_getc()

This function is used by the line driver to call PPP with characters received from the UART.

Syntax

```
int (*ln_getc)(int unit, int byte)
```

where:

unit Is the PPP unit number.

byte Is the data byte to be passed to the protocol stack from the UART.

Return value

Returns one of the following:

0 If the protocol accepted the byte.

nonzero If the protocol could not accept the byte.

Usage

Previous versions of the PPP stack required the line driver to call `ppp_input()` directly with each character received. From PPP Release 1.4 onwards, the approved technique is for the `ln_getc()` entry of the `com_line` structure to be initialized to point to the `ppp_input()` routine, and for the line driver to call the protocol input routine this way. The return value from the protocol stack is informational only. You do not have to take any action upon failure.

Your main loop (or a separate thread if you are using an RTOS) must regularly check for characters arriving at each unit, and then pass them to the PPP stack using `ln_getc()`.

3.3.3 ln_hangup()

Line drivers disconnect the line. On modems, this is a hang-up. On return, the line device must be ready to initiate another connection using `ln_connect()`.

Syntax

```
int (*ln_hangup)(int unit)
```

where:

unit Is the PPP unit number.

Return value

Returns one of the following:

0 If the hardware hang-up event had no errors.

nonzero error code

If the event had errors. This return is strictly for information purposes. PPP does not take any action based on the value returned.

3.3.4 ln_putc()

This function sends a byte on the line. If the line hardware is temporarily blocked (for example, full FIFO or XOFF state), the line driver must either block, or queue the byte for later transmission.

Syntax

```
int (*ln_putc)(int unit, int byte)
```

where:

unit Is the PPP unit number.

byte Is the byte to be transmitted.

Return value

Returns one of the following:

0 If *byte* was sent without error.

nonzero If an error occurred while trying to send *byte*. PPP assumes the link has failed, dumps the packet, and does not retry.

Indeterminate conditions, such as queuing a byte in a FIFO for sending, should return 0 unless a clear device failure is detected.

3.3.5 ln_speed()

This function queries the line speed and returns the bit rate. It is currently used only for informational purposes, such as SNMP queries on interface speed. On devices where it is difficult to obtain accurate speed information, it is acceptable to approximate. For example, a 28.8 modem might always return 28800. If the device is not connected, the maximum nominal speed of the device should be returned.

Syntax

```
long (*ln_speed)(int unit)
```

where:

unit Is the PPP unit number.

Return value

Returns the current speed (bits per second) of the line.

3.3.6 ln_state()

This function queries the line state to determine if the line is connected or not. A line is considered to be connected if it has completed an `ln_connect()` call without error, is currently working, and has not been disconnected with a call to `ln_hangup()`.

Syntax

```
int (*ln_state)(int unit)
```

where:

unit Is the PPP unit number.

Return value

Returns one of the following:

0 Connected and working.

1 Not connected, but may be connectable.

2 Broken, noncorrectable hardware error detected.

3 Dialing in progress.

3.3.7 ln_write()

This function sends a block of data on the line. This is currently unused by PPP, but is expected to be needed for future development. It could be implemented as a **for** loop that calls `ln_putc()` for each byte in the block.

Syntax

```
int (*ln_write)(int unit, char *block, int length)
```

where:

unit Is the PPP unit number.

block Is a pointer to the buffer containing the block of data to be sent.

length Is the number of bytes to send.

Return value

Returns one of the following:

0 If the *block* was sent without error.

nonzero If an error occurred while sending the *block*. PPP assumes the link has failed, dumps the packet, and does not retry.

Indeterminate conditions, such as queuing a byte in a FIFO for sending, must return 0 unless a clear device failure is detected.

3.4 PPP entry points

In addition to implementing the functions documented in *Serial line drivers* on page 3-10, you must make calls directly to the PPP software. These calls are documented in this section. The functions are as follows:

- *lcp_lowerdown()*
- *lcp_lowerup()* on page 3-19
- *ppp_input()* on page 3-20
- *ppp_timeisup()* on page 3-21
- *prep_ppp()* on page 3-22.

3.4.1 lcp_lowerdown()

This function informs the PPP stack that the communications line is no longer available and that the current connection must be tidied up and closed.

Syntax

```
void lcp_lowerdown(int unit)
```

where:

unit Is the PPP unit number.

Return value

None.

Usage

This function must be called from the line driver whenever a connected device terminates the connection. This includes terminations that are the result of an `ln_hangup()` request. The PPP layers might attempt to send bytes using `ln_putc()`. However, the line code is free to discard them.

Failure to call this function after unexpected disconnection usually results in PPP being unable to use the unit.

3.4.2 lcp_lowerup()

This function informs the PPP stack that the communications line has become available and that a new connection must be established.

Syntax

```
void lcp_lowerup(int unit)
```

where:

unit Is the PPP unit number.

Return value

None.

Usage

This function must be called from the line driver code when a change in line status from `not connected` to `connected` is detected. The most common example of this is a modem in auto-answer mode accepting an incoming call. This callback to PPP initiates the correct PPP events. In the example of a modem answering, the PPP layer sends packets to begin an LCP link as a server.

Note

- This call may take a long time to complete, so it must not be called:
  - from an *Interrupt Service Routine* (ISR)
  - while interrupts are disabled
  - from a time-critical section of code.
- Because an initial LCP configuration request (CONFREQ, the beginning of option negotiation) is sent in the context of this call, the line device must be prepared for a series of calls to `ln_putc()` before this function returns.

3.4.3 ppp_input()

This function must be called from your code whenever data is received from the UART while the PPP connection is active.

Syntax

```
void ppp_input(int unit, int c)
```

where:

unit Is the PPP unit number.

c Is a character received from the UART.

Return value

None.

Usage

The `ppp_input()` function must be called with the unit number of the receiving unit on which the character was received, and the received character.

Note

There can be a considerable amount of processing performed by `ppp_input()`, especially at the end of a frame, so it must not be called:

- from an ISR
- while interrupts are disabled
- from a time-critical section of code.

You can see an example of how to call `ppp_input()` in the modem dialer code supplied (`\modem\dialer.c`). See also *ln_getc()* on page 3-12.

3.4.4 ppp_timeisup()

This function drives the internal PPP timers.

Syntax

```
void ppp_timeisup(void)
```

Return value

None.

Usage

The `ppp_timeisup()` function is provided in `sys_np.c` and must be called every second by the system.

Note

Because the `ppp_timeisup()` function can perform a considerable amount of work, it must not be called:

- from an ISR
- while interrupts are disabled
- from a time-critical section of code.

Also, it is important not to call this function more or less than once per second, as this causes the PPP timers to expire at the wrong time.

3.4.5 prep_ppp()

This is the first PPP function called from the IP initialization logic.

Syntax

```
int prep_ppp(int firstnet)
```

where:

firstnet Is the index of the first interface to initialize for PPP.

Return value

Returns the index of the next available `nets[ ]` entry.

Usage

The `prep_ppp()` function must be called from `prep_ifaces()` in the `ipport.c` file to bind PPP to the TCP/IP stack. This function sets the number of interfaces (`nets[ ]`) to be used for PPP and maps one PPP unit (usually a serial link) to each interface. See the *Porting TCP/IP Programmer's Guide*.

Chapter 4
Modem Functions

The files in the `\modem` subdirectory are provided as an example of how to interface a Hayes-compatible modem to the PPP stack. The functions implemented within these files are described in this chapter.

The modem functions are accessed from the PPP stack using the `com_line` structure described in *Serial line drivers* on page 3-10. In turn, the modem functions make calls to the UART device driver in the `\integrator` directory. The UART driver is documented in the *Porting TCP/IP Programmer's Guide*.

This chapter contains the following sections:

4.1 dialer.c

The functions in `dialer.c` are the primary interface between the PPP stack and the modem. The `dialer.c` file contains functions to manage the dialing of connections, passing data between the modem and the PPP stack, and shutting down the PPP stack when a connection terminates.

The `dialer.c` file is provided as an example of how to interface between the PPP stack and a Hayes-compatible modem, and currently only supports one device.

The functions are listed in Table 4-1.The Interface column of the table shows the functions that form the interface between PPP and the modem.

Table 4-1 dialer.c functions

Function name and page reference	Interface
dial() on page 4-3	No
dial_check() on page 4-4	No
dialer_status() on page 4-5	No
modem_cmd() on page 4-6	No
modem_connect() on page 4-7	Yes
modem_getc() on page 4-8	No
modem_gets() on page 4-9	No
modem_hangup() on page 4-10	Yes
modem_init() on page 4-11	Yes
modem_lstate() on page 4-12	Yes
modem_putc() on page 4-13	Yes
modem_reset() on page 4-14	Yes
modem_speed() on page 4-15	Yes
modem_state() on page 4-16	No
modem_write() on page 4-17	Yes
modem_no_carrier() on page 4-18	No

4.1.1 dial()

This function starts the dialing process, the first step in establishing a connection.

Syntax

```
void dial(char *phone_num)
```

where:

phone_num Is a string containing the number to be dialed.

Return value

None.

Usage

Dialing can be initiated only in `Idle` or `Auto-Answer` mode. The `dial()` function sends the string `ATDT` to the modem, followed by the requested phone number, and a terminating carriage-return character, using the `modem_cmd()` function.

The `dial()` function does not wait for a connection to be established. It updates the modem status to indicate that dialing is in progress.

4.1.2 dial_check()

This function drives the state of the dialer.

Syntax

```
void dial_check(void)
```

Return value

None.

Usage

The `dial_check()` function must be called periodically by the operating system so the dialer can advance from state to state without blocking. In the example applications provided with the source code, `dial_check()` is called from the `tk_yield()` function.

While the modem is not connected, characters received from it are processed for responses such as `CONNECTED` or `BUSY`, and the state of the dialer is changed accordingly.

After the modem has entered the `CONNECTED` state, characters received from the modem are passed through to the PPP layer, the line is monitored for inactivity timeouts, and the DCD status is monitored. If DCD becomes low, the PPP layer is informed by calling `lcp_lowerdown()`.

4.1.3 dialer_status()

This function prints debug information about the status of the dialer.

Syntax

```
int dialer_status(void *pio)
```

where:

pio Is a pointer to the GenericIO structure where debug information is to be written.

Return value

Always returns 0.

Usage

The `dialer_status()` function calls `modem_portstat()` and `uart_stats()` to report on the status of the modem and the UART. This function can be used for debugging.

4.1.4 modem_cmd()

This function sends an AT command to the modem.

Syntax

```
int modem_cmd(char *data)
```

where:

data Is a null-terminated string containing the command to send to the modem.

Return value

Returns one of the following:

0 No error detected. The modem took and echoed the command.

–1 If an error occurred.

Usage

The modem_cmd() function attempts to send the command up to three times. It looks for the echo of the command from the modem as indication of success.

4.1.5 modem_connect()

This function checks that a connection to the remote host has been established and establishes one if it has not been done already.

Syntax

```
int modem_connect(int unit, struct com_line *lineptr)
```

where:

unit Is the interface unit number.

lineptr Is the pointer to the com_line structure for this unit.

Return value

Returns one of the following:

0 If the line is or was connected.

-1 If the line could not connect because of a temporary problem, for example, if the line is busy.

-2 If the line could not connect because of a hard error.

3 Dialing in progress.

Usage

The modem_connect() function is called from the send code of the PPP interface when a packet is to be sent. This checks that the modem and UART are ready and attempts to dial if they are not.

If the line is already connected, modem_connect() returns 0 (connected) immediately. Otherwise, modem_connect() calls dial() to send the dial string to the modem, and then returns the value 3 (dialing in progress). The underlying protocol that called modem_connect() re-calls it periodically until either 0 (connected) or one of the negative error values is returned.

4.1.6 modem_getc()

This function gets a character from the modem.

Syntax

```
int modem_getc(unsigned tmo)
```

where:

tmo Is the timeout, in seconds, to wait for modem characters.

Return value

Returns one of the following:

char Returns the next character from the modem, if one appears within the specified number of seconds.

–1 If timeout occurs without a character arriving from the modem.

Usage

The characters returned by this function are expected to be a reply to, or an echo of, a command. This function is not intended for general data stream gathering.

4.1.7 modem_gets()

This function gets a line of input from the modem.

Syntax

```
void modem_gets(int wait)
```

where:

wait Is the number of ticks to wait for input.

Return value

None.

Usage

This function is designed to get responses to commands that are sent to the modem. It returns immediately if the modem is currently connected and also if no input comes within *wait* ticks. You can specify the *wait* parameter as `0` for immediate return if input is not waiting.

The `modem_gets()` function leaves the input read from the modem in the `modem_in[]` buffer and sets `modem_index` to point to the next location in `modem_in[]` to be used.

If any input from the modem is received within *wait* ticks, the input string is considered to be complete when no more characters are received for a whole second.

4.1.8 modem_hangup()

This function hangs up the modem line immediately and terminates any connection.

Syntax

```
int modem_hangup(int unit)
```

where:

unit Is the interface unit number.

Return value

Returns one of the following:

0 If successful.

–1 If not successful.

Usage

If PPP is being used, `modem_hangup()` calls `pppclose()` and `lcp_lowerdown()` to indicate that the line is no longer available. It then calls `modem_reset()` to hangup the phone line and re-initialize the modem.

4.1.9 modem_init()

This function sets up the modem.

Syntax

```
int modem_init(int unit)
```

where:

unit Is the interface unit number.

Return value

Returns one of the following:

0 If successful.

–1 If not successful.

Usage

The `modem_init()` function is called from `ppp_port_init` in the `ppp_port.c` file. It initializes the UART, resets the modem, and sends the initialization string to the modem.

4.1.10 modem_lstate()

This function reports which state the modem is in, as a numeric code.

Syntax

```
int modem_lstate(int unit)
```

where:

unit Is the interface unit number.

Return value

Returns one of the following:

0 Connected.

1 Not connected, but working.

2 Not connected. Hard error.

3 Dialing in progress.

Usage

The `modem_lstate()` function is used by the PPP stack to obtain the current state of the modem.

4.1.11 modem_putc()

This function sends a character through the modem connection.

Syntax

```
int modem_putc(int unit, int bByte)
```

where:

unit Is the interface unit number through which the character will be sent.

bByte Is the character to be sent.

Return value

Returns one of the following:

0 If successful.

–1 If not successful (timeout occurred).

Usage

The `modem_putc()` function is called by the protocol code to send single bytes. It must not be called unless the modem is in the connected state.

The `modem_putc()` function waits for space to become available for up to one second by calling `tk_yield()` in a loop and polling the UART using the `uart_ready()` function. When space is available, `modem_putc()` calls `uart_putc()` to actually send the character to the modem.

4.1.12 modem_reset()

This function resets the modem.

Syntax

```
void modem_reset(int unit)
```

where:

unit Is the interface unit number.

Return value

None.

Usage

The `modem_reset()` function is called to re-initialize the dial and initialization strings for the modem from the values read from NVRAM, and calls `uart_reset()` to reset the modem hardware.

4.1.13 modem_speed()

This function returns the bit rate value extracted from the last `CONNECT` string.

Syntax

```
long modem_speed(int unit)
```

where:

unit Is the interface unit number.

Return value

Returns the last known bit rate (bits per second).

Note

The return value might not relate to the actual transmission speed between this modem and the remote unit at all. Some modems report the connection speed between the UART and the modem, rather than the line speed between the modems.

4.1.14 modem_state()

This function obtains a string describing the state of the modem.

Syntax

```
char *modem_state()
```

Return value

Returns a pointer to a static read-only string containing a description of the current status.

4.1.15 modem_write()

This function sends several characters to the modem.

Syntax

```
int modem_write(int unit, char *buf, int len)
```

where:

unit Is the interface unit number.

buf Is the pointer to the characters to be sent.

len Is the number of characters to send.

Return value

Returns one of the following:

0 If successful.

–1 If timeout occurs while waiting to send.

Usage

The `modem_write()` function sends *len* characters of data from *buf* to the modem. This function is not actually used by the current PPP stack. It is included to accommodate future releases of PPP.

4.1.16 modem_no_carrier()

This function determines whether the modem has output a `NO CARRIER` string.

Syntax

```
int modem_no_carrier(int c)
```

where:

c Is a character received from the modem.

Return value

Returns one of the following:

TRUE When `NO CARRIER` is found in a sequence of characters.

FALSE Otherwise.

Usage

The `modem_no_carrier()` function only needs to be used if the modem does not support the DCD line.

4.2 login.c

The functions in `login.c` implement a login script mechanism that you can use to negotiate through the **login:** and **password:** prompts issued by some dial-in servers. The functions are:

- *do_script()* on page 4-20
- *login()* on page 4-21
- *log_input()* on page 4-22
- *log_output()* on page 4-23
- *logserver()* on page 4-24.

4.2.1 do_script()

This function processes a login script for either `login()` or `logserver()`.

Syntax

```
int do_script(char *sfilename)
```

where:

sfilename is the filename from which to read the script.

Return value

Returns one of the following:

0 If successful.

–1 If not successful.

Usage

The `do_script()` function is an engine to open a text file and treat it as a script for logging into a remote host that has just been dialed into. The script file provides a user-configurable script to log in to various hosts. It contains text for strings to output to the modem and text for expected replies from the modem, including timeouts.

Special characters (when at the start of a line) are:

`#` A comment character.

Recognized commands are:

`input` *secs string*

Gets characters from the modem until the *string* is matched or *secs* seconds have elapsed. If the timeout expires before the *string* is found, the script is aborted.

`output` *string*

Sends a *string* to the modem. It can contain escape sequences.

`echo` *text*

Sends *text* to the user console (or log).

4.2.2 login()

This function executes a login script to log in to a remote host after the connection has been started.

Syntax

```
void login(void)
```

Return value

None.

Usage

This is called by the link control code, for example, the dialer, when a link connection has been established. If no login script has been specified, it marks the connection as logged in and returns. Otherwise, it executes the login script.

The `login()` function sets `loggedin` to `TRUE` if the login script was successful (or did not exist), or `FALSE` if the login attempt failed.

4.2.3 log_input()

This function gets a string from the modem and checks that it matches the passed string.

Syntax

```
int log_input(char *string, int secs)
```

where:

string Is the sequence to look for.

secs Is the maximum time, in seconds, to wait for the sequence.

Return value

Returns one of the following:

TRUE If *string* was seen within *secs* seconds.

FALSE If *string* was not seen within *secs* seconds.

Usage

The `log_input()` function waits for up to *secs* seconds to see if *string* arrives from the modem. It leaves the input from the modem in `modem_in[ ]`.

4.2.4 log_output()

This function sends a string to the modem, converting escape sequences.

Syntax

```
void log_output(char *string)
```

where:

string Is the string to send.

Return value

None.

Usage

The `log_output()` function sends the printable ASCII characters from *string* to the modem, converting escape sequences to their corresponding control characters. The `log_output()` function waits for space to become available before sending each character.

The `log_output()` function converts the following escape sequences:

`\n` Newline character (012)

`\r` Carriage return (015)

`\t` Tab (011)

`\x` Character *x* (for example, \\ to get a single backslash).

4.2.5 logserver()

This function executes a login script to allow a remote host to log into this system when they dial in.

Syntax

```
void logserver(void)
```

Return value

None.

Usage

The `logserver()` function is called by `dial_check()` in `dialer.c` to handle logging on users who dial in. If the log server file has been specified in the NVRAM, the `logserver()` function runs that script and sets `loggedin` to `TRUE` or `FALSE`, accordingly. If no log server file has been specified, `logserver()` sets `loggedin` to `TRUE`.

This function leaves `loggedin` set to `TRUE` or `FALSE`.

4.3 mdmport.c

The functions in `mdmport.c` are the glue layer functions that are specific to porting the modem control module to other platforms. The functions are:

- *dial_delay()* on page 4-26
- *hangup()* on page 4-27
- *modem_clr_dtr() and modem_set_dtr()* on page 4-28
- *modem_DCD()* on page 4-29
- *modem_portstat()* on page 4-30.

4.3.1 dial_delay()

This function causes a delay by allowing the task loop to spin.

Syntax

```
void dial_delay(unsigned long ticks)
```

where:

ticks Is the number of clock ticks to delay for.

Return value

None.

Usage

The `dial_delay()` function is called to allow the task loop to spin for *ticks* clock ticks without re-entering dialer code. This function calls `tk_yield()` repeatedly until the requisite number of clock ticks have passed.

4.3.2 hangup()

This function hangs up the modem (used by the menus demonstration application, as documented in the *Command-line Interface Reference Guide*).

Syntax

```
int hangup(void *pio)
```

where:

pio Is a pointer to the GenericIO structure where debug information is to be written.

Return value

The `hangup()` function returns the same values as `modem_hangup()` (see *modem_hangup()* on page 4-10).

Usage

The `hangup()` function is used by the menus demonstration application to force a hangup of the modem. It calls `modem_hangup()` with the default modem unit number.

4.3.3 modem_clr_dtr() and modem_set_dtr()

These functions control the *Data Terminal Ready* (DTR) signal to the modem.

Syntax

```
void modem_clr_dtr(void)
```

```
void modem_set_dtr(void)
```

Return value

None.

Usage

The `modem_clr_dtr()` function drops the DTR signal to the modem, and the `modem_set_dtr()` function is used to assert the DTR signal. Dropping the DTR signal usually causes the modem to hang-up the line, if it is connected, and to return to command mode when the DTR signal is asserted again.

4.3.4 modem_DCD()

This tests the current status of the *Data Carrier Detect* (DCD) input from the modem.

Syntax

```
int modem_DCD(int unit)
```

where:

unit Is the interface unit number.

Return value

Returns one of the following:

TRUE If the carrier detect is down.

FALSE If the carrier detect is up.

Usage

The `modem_DCD()` function is called by the dialer functions to determine the current state of the DCD input from the modem.

4.3.5 modem_portstat()

This prints statistics about the modem port.

Syntax

```
void modem_portstat(void* pio int unit)
```

where:

pio Is a pointer to the GenericIO structure where debug information is to be written.

Return value

None.

Usage

The `modem_portstat()` function is used for debugging purposes to print out statistics about this modem port.

Appendix A
Testing the PPP stack

A complete guide to dial-up networking on Windows NT is beyond the scope of this document. However, this Appendix gives a checklist of points to remember when testing the ARM PPP stack against a Windows NT machine. This information is written for Windows NT Workstation 4, but NT Server 4 should be almost identical. This Appendix contains the following sections:

A.1 Setting up the PC

Use the following checklist to set up your PC:

1. The PC must be set up to use TCP/IP.
2. Enable IP forwarding, if necessary:
   - **Control Panel → Network → Protocols → TCP/IP → Routing**
3. Connect a crossover (*null-modem*) serial cable between Serial B on the Integrator board and a spare COM port on the PC.
4. Ensure that Remote Access Service (RAS) is present:
   - **Control Panel → Network → Services**
5. Configure RAS to both dial out and receive calls:
   - As above, then **Properties → Configure**
6. Configure RAS to allocate IP addresses appropriate for your environment:
   - As above, but click on **Network** instead of **Configure**, then the **Configure** button next to **TCP/IP**.
7. Reboot when prompted.
8. Add a modem of type *Dial-Up Networking Serial Cable*, attached to the appropriate COM port. You might be prompted to do this during the installation of RAS, above.
9. Configure the modem to operate at 115200 bits/second, and to use hardware flow control.
10. Add a local user account:
    - **Start → Programs → Administrative Tools → User Manager**
    - Set up the account with the name PPP and password `letmein` (this is the string specified after `CHAP secret` in the file `direct.nv` of the application being tested).
    - Give the user dial-in permission (click on the **Dialin** button at the bottom of the **User Properties** dialog box).
11. Check that the RAS services have started properly (**Control Panel → Services**), set them to Automatic startup, and start them if necessary.

A.2 Build and networking considerations

For the build:

1. Ensure that a PPP target is built, and that DIRECT_RAS is defined in ipport.h.
2. Check the settings in `direct.nv`.

For the network:

1. Consider how routing is going to work between the Integrator board and any other hosts it needs to access.
2. Choose IP numbers to assign to the PPP link.
3. Set up any routes necessary on other hosts. For example, the SMTP server (for the maildemo project) needs a route to the Integrator board specifying the NT machine as the gateway.

A.3 Connecting to the Integrator board

Use the following checklist when connecting to your Integrator board:

1. Some of the PPP demonstration programs operate as *servers* (that is, the PC calls them), some as *clients* (they call the PC), and some can operate in both modes.
2. Add a phonebook entry to dial the Integrator board:
   - **My Computer → Dial-Up Networking**

   The phone number field is not used.
3. Use Dial-Up Networking to make a connection to the Integrator board when a program is running as a *server*.
4. Monitor the progress of an incoming *client* connection using the Dial-Up Networking Monitor:
   - **My Computer → Dial-Up Networking → More → Monitor status**
5. Check how NT is routing packets by using the route command. At a command prompt, type:

   ```
   route print
   ```

   See *Routing* on page A-5 for more details.

A.4 Routing

Sometimes, NT adds an incorrect route for an incoming PPP connection. If this is the case, delete it and add a correct one.

For example, if the Integrator board is using 192.168.168.2, and the PC is using 192.168.168.1, there must be a route to 192.168.168.2 using 192.168.168.1 as a gateway.

The following commands are useful in this case:

```
route print
route delete 192.168.168.2
route add 192.168.168.2 gw 192.168.168.1
```

Glossary

ADS	ARM Developer Suite.
CHAP	Challenge-Handshake Authentication Protocol.
DCD	Data Carrier Detect.
DHCP	Dynamic Host Configuration Protocol.
DTR	Data Terminal Ready.
FIFO	First In, First Out.
FSM	Finite State Machine.
IPCP	Internet Protocol Control Protocol.
ISDN	Integrated Services Digital Network.
ISR	Interrupt Service Routine.
LCP	Link Control Protocol.
MD4	Message Digest 4.
MD5	Message Digest 5.
NVRAM	Non-volatile Random Access memory.
PPP	Point-to-Point Protocol.

RAS Remote Access Service.

RTOS Real-time Operating System.

SMTP Simple Mail Transfer Protocol.

SNMP Simple Network Management Protocol.

TCP/IP Transmission Control Protocol/Internet Protocol.

UART Universal Asynchronous Receiver/Transmitter.

UDP User Datagram Protocol.

UPAP User/Password Authentication Protocol.

VJC Van Jacobson Compression.

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.

S

T

U

V

Y

Symbols