Point-To-Point Protocol Technical Reference

Interniche Legacy Document

Version 1.00

Date: 18-May-2017 11:10

Table of Contents

1 Introduction

This document is a "how-to" manual to enable an embedded systems engineer to use the InterNiche PPP code in an embedded product. The engineer should be expert in the C programming language and have a good knowledge of embedded systems. A conceptual understanding of what PPP does and familiarity with networking concepts is also helpful.

If you will be using InterNiche PPP with an InterNiche TCP/IP stack and OS, and are targeting a hardware platform supported by InterNiche, then there is very little work involved. You simply need to select which PPP options you want in you product (see PPP Options) and compile to get PPP working on your development target. To support hardware which is not directly supported by InterNiche you will also need to write a UART driver or other "line driver" for your device; see Line Drivers.

This document also covers the basics of moving the InterNiche PPP to a non-InterNiche IP stack. However this is considerably more complex than using InterNiche's IP. The porting process can take anywhere from a few hours to several days.

1.1 Terms and Conventions

In this document, the term "PPP", when used without other qualification, means the InterNiche PPP code as ported to an embedded system. "System" refers to your embedded target system. A "user" or "porting engineer" usually refers to the engineer who is porting the PPP. An "end user" refers to the person who ultimately ends up using the "user's" product.

Names of files, C structures and C routines are displayed as follows: c_routine()

Source from C programs is displayed in these boxes:

```
/* C source file - the world's 1 millionth hello program. */
main(){
   printf("hello world.\n");
}
```

Client and Server

The terms "client" and "server" are used throughout this manual and the PPP code to distinguish the PPP node which initiates the connection (the client) from the one which receives it (the server). For example, when a PC user dials into an ISP and the ISP's machine answers the telephone call, the user is the client and the ISP is the server. The InterNiche stack can function as both a client and a server.

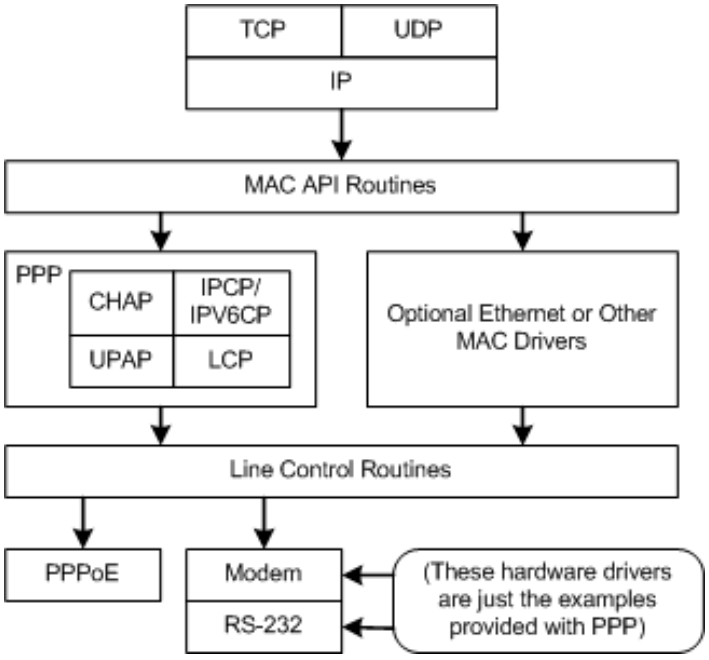
1.2 PPP Description

PPP is a specification for the transmission of network data over serial lines or serial line-like devices. At a minimum it provides the following functionality:

- Converts the 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
- Does compression of TCP/IP protocol headers
- Verifies the identity (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 InterNiche provides additional software with PPP that does this for a standard Hayes command set modem. The InterNiche PPP provides software for the two Network Control Protocols (NCPs) used to transfer IP packets over PPP (IPCP and IP6CP) but does not include support for non-IP protocols such as AppleTalk or DECnet.

This graphic shows a fairly complex example of how PPP might fit between the IP protocol family and the hardware links. In this case the line hardware can be either a modem or PPPoE (PPP-over-Ethernet). Note that CHAP, UPAP, LCP and IPCP/IP6CP are part of the PPP box.



PPP is actually a family of protocols, all working together to provide the services described above. Some members of the family (LCP, IPCP, IP6CP) provide a virtual connection service and handle a set of options, such as which authentication to use, whether to compress packets, etc. Each connection protocol moves these connections amongst states as described by the FINITE State Machine (FSM) definition in the PPP specification (RFC1661). Other members of the PPP family provide services to the connection protocols, such as security (CHAP, PPP), and connection state management (FSM).

The InterNiche package implements all of the protocols required for IP transmission, as well as optional protocols for authentication and TCP/IP header compression. It does not include protocols to upport non-IP networks, such as DECnet and AppleTalk. The required hooks for these protocols exist in the InterNiche PPP, but the protocols themselves are not included. A brief description of each layer follows.

IPCP and P6CP - Network Control Protocols

IPCP and IP6CP are two separate protocols for transferring IPv4 and IPv6 packets over PPP. All packets sent or received on the PPP connection are encapsulated in one of these two protocols. IPCP and IP6CP are defined by different RFCs and have entirely different sets of options that must be negotiated before data can be transferred over a PPP connection.

When a PPP session is established, the LCP layer negotiates whether the sessions will use IPv4 or IPv6. Different PPP links can use different IP protocols simultaneously.

CHAP - Challenge Handshake Authentication Protocol

CHAP is the primary mechanism for a PPP node to guarantee that the host on the other end of the line is who it says it is. This is initiated by sending a CHAP message (the challenge) via LCP from one PPP host to the other. The CHAP challenge contains a digest generated by the industry standard MD5 digest algorithm based on an ASCII string (called a secret) which is know to both hosts. The challenged host must then send back the correct CHAP reply, or the connection is terminated by the challenger. Generally either host may send the CHAP challenge at any time after the LCP connection is established.

The RSA MD5 message digest algorithm provides an extremely high level of security, however the LCP specification provides a "digest byte" in case another algorithm is ever needed.

UPAP - User/Password Authentication Protocol

UPAP is similar to CHAP except that a user-name and password are used to generate the authentication packets. UPAP, also called just plain PAP, does not compute a digest with the password information as CHAP does. This mechanism is less secure but requires less memory and CPU overhead. It is also generally easier to implement and debug.

FSM - Finite State Machine

This is not an actual protocol, but rather the definitions for the series of events that a PPP connection protocol moves through, from the initiation of the link to termination. As required by the specifications, LCP, IPCP, and IP6CP use the same state machine. That is, all three protocols use the same states, and they transition between states in exactly the same manner. However, each protocol has an entirely different set of routines to implement these states and transitions.

PPPoE - PPP over Ethernet

PPP over Ethernet is a standard (RFC2516) method of transferring PPP packets over Ethernet hardware. Ethernet differs from line oriented PPP hardware in that it is "multidrop". That means it may have more than two computers attached to a line. On such media, a method is required to indicate which of the other computers on the network should receive each PPP packet. PPPoE provides this method.

1.3 The Reference Implementation

If you bought PPP with InterNiche's IP stack, it is provided with at least one Reference Implementation. Usually there will be an implementation of an embedded target development board similar to yours - you should request one from your InterNiche salesperson. There may also be a sample program that implements an entire embedded stack as a process on MS-Windows. The reference programs can establish PPP connections on standard Hayes dialup modems or PPPoE (PPP over Ethernet) and transfer packets. At a minimum, you will be able to ping hosts via PPP. If you have access to the Internet (for example through a commercial service provider) which supports PPP, you will be able to dial into the Internet with this reference program.

The Windows reference code will compile with Microsoft C. Other targets compile with tool sets appropriate for the target processor. In each case, the tool sets and devices used are described in a readme.txt file that comes with the demo software. Unless you are quite familiar with TCP/IP and PPP, and are comfortable working with complex networking code, we recommend you compile the reference program and experiment with it before starting your port. Instructions for this are in your TCP/IP Technical Reference Guide.

2 System Requirements

The InterNiche PPP software requires some support from the host system to operate. These needs fall into the categories listed below which are explained in detail in the sections that follow.

- Line Management Functions
- Static Memory
- Dynamic Memory - malloc() and free()
- Periodic Clock Tick

Most InterNiche customers receive the PPP code already implemented into a target system, and integrated into the InterNiche TCP/IP code and an RTOS. These customers don't need to worry about implementing the macros and services described in this section.

2.1 Line Management Functions

PPP needs to be able to send and receive characters on the target system's line hardware. It may also need to initiate a connection (i.e. dial the telephone number) or disconnect. To facilitate this, the porting programmer needs to provide a set of low-level routines. If more than one type of line is to be used, for example both DSL and modems, then a set of routines need to be provided for each type.

The PPP code accesses these line functions by defining a structure that contains a set of pointers to the line's routines. The porting programmer should make certain that all these pointers are set to the appropriate functions at system initialization time, even if the routine simply returns. Providing these routines is generally the bulk of the work when implementing PPP on a new target system.

The PPP code comes with two sets of line management functions: A PC serial line ("COM" port), and a loopback driver. If your target hardware is an embedded PC, and you intend to use an ordinary modem, then you can use the COM port line drivers exactly as provided. The loopback drivers are for testing only and are not expected to be the primary line drivers of a real product. Line drivers are also provided for many standard embedded target systems. Contact InterNiche to find out if we have drivers for you target system.

The line driver calls are described in detail in Line Driver Calls.

2.2 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 may be moved to RAM at boot time. The exact amount of code space required will vary depending on which PPP optional features you enable through #defines, your processor, and your compiler.

2.3 Dynamic Memory

PPP allocates a single control structure for each connection. It may also allocate additional blocks of memory for PPPoE control structures. PPP allocates these areas by calling functions which 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 only used for only one type of buffer or structure. This allows the macros to be mapped to a "partition" based heap manager with no wasted memory.

These functions are described in detail in Porting Programmer Provided Routines.

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

If your system does not support calloc(), or you don't want to use it for performance reasons, then there is another easy way to implement these memory functions. You can reserve arrays of static buffers of the sizes required, and return pointers them from the allocation macros. The exact sizes required will vary with the environment (CPU type, compiler packing options, etc.) so you should use sizeof() operators in your static declaration statements.

2.4 The Clock Tick

The PPP code includes a routine that should be called by the system once a second. This routine drives the re-transmissions and time-outs.

In addition, the PPP code expects the system to maintain a clock tick counter variable named cticks, that is incremented TPS times a second. The macro TPS should be defined in your ppp_port.h file or one of its nested includes. On the Windows reference package, cticks is incremented 18 times a second, and so TPS is defined to be 18.

3 Architectural Overview

This section describes the organization of the PPP code and the main internal structures. It is for informational purposes only. As noted elsewhere, porting engineers should not modify any files listed as portable in section 4 without first consulting HCC.

Most of the PPP code is organized into files which bear the name of the layer or module implemented. For example lcp.c implements the LCP functionality. The FSM and system code are implemented in pppfsm.c, and code related to PPP interactions with the IP stack and RTOS is in pppsys.c. Most of the PPP definitions that are not specific to a single layer are in the header file `ppp.h`. Each of the files is described in detail starting in section 4.

The connection-oriented modules (IPCP, LCP) have a number of functions that handle the FSM events for the module. These are table driven, and thus each module has a prot_funcs structure as defined in `ppp.h`.

PPP ports to target systems not directly supported by InterNiche may require some changes to the PPP code. PPP is designed so all the changes will be in the files: `ppp_port.c`, `ppp_port.h`, and `poe_port.c` if you have PPPoE.

3.1 The mppp Structure

Each PPP connection is controlled by an M_PPP structure, which is defined in `ppp.h`. These are typically named mppp in the source code and referred to as "mppp"s in this manual. The mppp may contain sub-structures for CHAP, PAP, VJ authentication and other protocols that are compiled in the system. Since all these substructures (and some protocol related variables) are "ifdef-ed", the size of the mppp varies greatly from port to port. The PPP code maintains a master list of mppp structures, pointed to by the variable ppp_list and maintained by the code in ifppp.c. In a typical embedded system where PPP supports just one modem or UART, there will be only a single mppp.

3.2 Line Drivers

InterNiche PPP accesses the underlying network hardware by means of the line driver API. Each type of device (UART, Modem, PPPoE) which is to carry PPP traffic must implement these calls. The calls are described in detail in the section starting in Line Driver Calls. InterNiche provides these calls for several common embedded hardware development systems, including Hayes™ "AT" command modems over several common UARTs.

Line drivers often do more than just perform IO to UARTs. They are also responsible for connecting "part time" links such as modems, and signaling the PPP code whenever a link comes online. In order to support Hayes compatible Modem dialing, a modem line driver is provided that assumes a simple byte oriented UART driver beneath it. Similarly, the entire PPPoE protocol is implemented as a line driver.

com_line Structures

Line drivers are managed by the structure com_line, which is defined in the header file comline.h. This file is shipped in the h files directory since it is referenced by many InterNiche modules other than PPP. Each mppp structure contains a com_line structure. The com_line structure actually describes the interface between an upper layer (in this case usually PPP) and a lower layer, such as modem code. Note that another com_line structure may exist below the lower layer, for example to map a modem dialer to a UART driver. The line structure contains pointers to the lower layer's entry point routines, as well as type and session information about both the upper and lower layers. The types for both layers are given by one of the LN_ types defined in comport.h. These types can be used by porting engineers to control the types assign to new PPP connections as they are created. The predefined line types are:

```
#define LN_PPP       1     /* upper layer is PPP */
#define LN_SLIP      2     /* upper layer is SLIP */
#define LN_UART      3     /* lower layer is a UART */
#define LN_ATMODEM   4     /* upper/lower layer is a modem */
#define LN_PPPOE     5     /* lower layer is PPPOE */
#define LN_LBXOVER   6     /* lower is loopback crossover (for test) */
#define LN_PORTSET   7     /* (init) lower will be set by callback */
```

Of special interest is the LN_PORTSET type; this indicates that the line is to have a special type defined by the port. Setting Line Types contains more information about line types and how to assign them.

3.3 The PPP Finite State Machine - FSM

The FSM is implemented as a table in the file pppfsm.c. The entries in this table match the PPP FSM table in RFC1661. On each of the events described in the RFC (such as receiving a control packet, a line becoming connected, or shutting down), a call is made to ppp_fsm() with the mppp, protocol (LCP, IPCP or IP6CP) and event passed as parameters. The code in ppp_fsm() looks up the event in the table based on the current connection state and performs the appropriate actions. Most of RFC1661 is devoted to describing the events and actions of the state machine.

The FSM lowerup and lowerdown events are initiated outside the PPP code. The entry points `ppp_lowerup()` and `ppp_lowerdown()` are provided to implement the FSM transitions required by the external events. Line drivers should call these functions as they detect lines connecting or disconnecting.

4 Porting Step By Step

This section outlines the steps needed to port the InterNiche PPP code into the software of a pre-existing InterNiche IP stack. If you are implementing PPP at the same time as the IP stack, simply defining USE_PPP in `ipport.h` and including the PPP sources in your makefile or build script will do most of the work for you. In any case, porting engineers should only modify the files listed in the table of Reference Port Files. All other files are considered portable" (see next section) and should not need to be changed.

Generally, it should not be necessary for the porting engineer to modify the portable files.

If, during the course of a routine port, the porting engineer determines that modifications to portable files appear necessary, (s)he should FIRST discuss the intended modifications with InterNiche technical support staff who will either suggest an alternative or will provide modified source files to reflect a necessary change.

Porting programmers may want to review the list of function calls in the next section. At a minimum you will need to ensure that prep_ppp() and ppp_timeisup() get called as appropriate, that the allocation routines are properly mapped, and that a line driver is available.

Porting programmers who use the InterNiche IP stack may simple rely on the InterNiche stack to initialize the driver and PPP code. Other IP stacks will need a glue layer which maps their initialization, send and close calls to PPP. PPP's calls are those described for an InterNiche Net Structure as described at length in the InterNiche TCP/IP Technical Reference Manual.

PPP can be customized for your particular application in a variety of areas. This includes behavior such as verifying user-names, connection management (dial and hangup), which IP addresses to use, and PPPoE options. Usually this involves writing a simple C routine to implement the desired behavior for your system. The routines which handle these functions are described starting in Customizing Your PPP Port. In each case, a reference implementation is provided with the PPP code as delivered. The routines for verifying a user-name and getting a CHAP secret are contained in the file ppp_port.c. Routines specific to PPPoE are in the file poe_port.c. Other routines are implemented via pointers to functions - if the pointer is NULL the function is not called, otherwise the pointer is assumed to point to code.

4.1 Source Files

As provided, the PPP source code is several .c source files and some .h (include) files. These are called the PPP "portable" or "port-independent" source files, and should not need to be modified for a simple PPP port. The Reference Port Files are provided as part of the reference package, and implement the port dependent portions. The functionality of these files will need to be duplicated in the target system as part of the porting process.

The portable .c source files are:

File	Description
pppfsm.c	PPP FSM management (core PPP code)
pppsys.c	PPP interface with the system
ifppp.c	InterNiche MAC interface entry points for PPP
pppchar.c	support for HDLC-like encapsulation
ppp_mod.c	Contains the ppp_module definition and the module's "prep" and "close" functions
pppoe.c	PPP over Ethernet support (product option)
ppp_dhcp	Support DHCP client over PPP links
ppp_loop.c	PPP loopback test line driver
lcp.c	LCP layer
ipcp.c	IPCP layer
ipv6cp.c	IP6CP layer
vjcomp.c	Van Jacobson compression
chap.c	CHAP Authentication protocol
upap.c	PAP Authentication protocol

The portable header files are:

File	Description
ppp.h	core PPP definitions - connection object, etc.
lcp.h	LCP protocols definitions
ipcp.h	IPCP protocols definitions
ipcp6.h	IP6CP protocols definitions
chap.h	CHAP protocols definitions
upap.h	PAP protocols definitions
ppp_loop.h	loopback test driver definitions
vjcomp.h	Van Jacobson compression
pppoe.h	PPPOE protocols definitions

The reference port files are both quite small - only a few K bytes of commented source. They are:

File	Description
inppp.c	This file contains port-dependent PPP functions and is found in the target-specific directory.
ppp_port.c	Generic PPP configuration. Found in the target-specific directory.
poe_port.c	PPP over Ethernet configuration and options
ppp_port.h	PPP compile-time defined options
ppp_nt.c	Code for the menu functions and their parameter definitions. These may be modified by the porting engineer.

The best first step is usually to compile the code as received from InterNiche. A readme.txt file your target directory (usually named after your development board) will explain what compiler and build tools to use for this. This will give you some hands on experience with PPP, and you will have the opportunity to step through the PPP code under a source level debugger. In the event something breaks during your port to the target machine, you will have a working reference platform to aid in debugging.

4.2 PPP Options

Early in the porting process you should decide which of PPP's optional features you want to utilize and set the ifdefs for them. These ifdefs are generally set in your ipport.h file, which controls all InterNiche compile-time options. The options they include are usually useful, but they can nearly double the size of the PPP code, so engineers of systems with limited memory may desire to omit some of them. Most ports can simply #define PPP_VJC and CHAP_SUPPORT, but the porting engineer should check the "ipport.h" file for other compile-time options which can be enabled.

C code excerpt showing several of the most commonly used options enabled:

```
#define PPP_VJC         1  /* VJ header compression */
#define CHAP_SUPPORT    1  /* CHAP authentication */
#define PAP_SUPPORT     1  /* Password authentication */
#define LB_XOVER        1  /* cross 2 loopback lines for test */
#define USE_PPPOE       1  /* cross 2 loopback lines for test */
#define PPP_MENUS       1  /* PPP - include the PPP debug menu */
#define PPP_DHCP_CLIENT 1  /* PPP - get IP address via DHCP */
#define PPP_CHARIO      1  /* PPP - support character read/write routines */
#define PPP_LOGFILE     1  /* PPP - ConPrintf() log to file option */
#define PPP_DNS         1  /* exchange DNS information in IPCP */
```

Keep in mind that PPPoE is optional. Don't define these if you don't have them!

Each of these compile switches is described below.

PPP_VJC

Enable the use of "Van Jacobson Compression" to compress TCP/IP headers. VJ Compression is a simple compression algorithm for TCP/IP headers that is named for the programmer who developed it. It is based on the principle that most of the information in the 40 byte TCP and IP headers doesn't 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. Since many TCP/IP packets contain only the headers, this can reduce the byte traffic on a PPP link by over 50% before any sort of data compression is applied to the data portion of the packet.

Disabling VJC should not prevent your system from inter-operating with any other PPP, it will just cause both systems to disable the feature and will typically result in lower throughput.

CHAP_SUPPORT

This includes the code for CHAP and MD5. For CHAP to actually be used, it must be configured with a secret by the end user and negotiated when LCP connects. If this feature is disabled then the porting programmer does not need to provide the get_secret() routine.

PAP_SUPPORT

Includes the code for UPAP. For UPAP to actually be used, it must be configured by the end user and negotiated when LCP connects.

USE_PPPOE

This includes the hooks in the regular PPP code (as well as our IP and packet handling modules) to support PPPoE. The entire contents for the pppoe.c file is also `#ifdef`-ed with this, so all trace of PPPoE is removed from builds not defining this option.

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 connections to each other. Bytes sent on either unit will be received on the other "crossed over" unit. This provides a useful testing environment for emulating PPP client/server conditions. This option is usually not needed in the final product; it is for development and testing only. The porting programmer may have to define the two unit numbers to be connected.

PPP_DNS

This includes code to exchange domain name server addresses as a part of the PPP IP Control Protocol (IPCP) option negotiation. This exchange also requires configuration on the part of the porting engineer and /or the end user. See the description of ppp_portlinksetup() for information on how to perform this configuration.

PPP_CHARIO

This includes code to perform the HDLC-like encoding and decoding of PPP datagrams. If you are using InterNiche TCP/IP support for devices like serial ports (#define USE_COMPORT) and modems (#define USE_MODEM) in ipport.h then this option is automatically enabled by the default ppp_port.h file. Otherwise you will need to make sure this is defined is you want to use a byte-oriented IO device for PPP.

PPP_LOGFILE

This includes the support for writing PPP log messages to a file, as well as to the system console. This should be used cautiously since many embedded systems do not have a file system, or want to use what file capacity they have sparingly. See Message Logging (ConPrintf).

PPP_MENUS

This compiles code to implement an InterNiche CLI extension for PPP. The menus include entries to create new connections, terminate existing ones, and alter configurations on the fly. The PPP menu system is described in detail starting in PPP Menu Options.

4.3 External Routines

The PPP code assumes some routines are available in the host system. The porting programmer will need to make sure all these are available to link PPP into his system image. Most of these are standard ANSI C library routines that are usually available with your compiler. If not, they can be easily provided.

The standard library routines used are listed here. Note that the mem routines are coded in UPPERCASE in the source code. This is an InterNiche convention so that the standard mem routines can be mapped to special high-speed versions in systems where the C library does not provide a well optimized **memcpy()** routine.

memcpy()	copy a memory block.
memcmp()	compare memory block.
strcpy()	copy a C string.

Additionally, the external checksum routine from the IP stack is only needed for VJ compression. A C language version is provided, and optimized assembly language routines are available for most processors.

cksum()	IP checksum routine>

4.4 Customizing Your PPP Port

The file `ppp_port.h` in the top-level target directory contains many important definitions that control both the ppp and pppoe compile-time configuration. Most of these parameters have default settings that will work well on most systems. Others such as `CHAP_SECRET_STRING` and the PPPOE access concentrator name should be modified by the porting engineer. The porting engineer should examine each variable in this file to ensure that it is set to the desired value.

Once you have the PPP code compiling, you may to add the hooks to modify PPP for your application. This section describes some areas that can be customized and their associated routines.

Message Logging

Message logging should not be lightly ignored on any PPP system, even embedded ones. For many reasons, to do so is to risk making an unusable product. The most common underlying link used with PPP is the inherently unreliable dialup modem. Further, there are so many PPP options that interoperability between any two systems is not guaranteed. This is explicitly mentioned in RFC1661. Finally, the single most common cause of PPP connections failing is incorrect user/password information.

When a PPP link fails, the user needs some way of finding out what the problem is. An unreliable modem or ISP may need to be replaced, and an inaccurate password will not fix itself. For all these reasons, your PPP device should have a way of communicating problems to human users. The traditional way to do this on desktop systems is implement a PPP message log file. The PPP code writes brief text messages to this file as connections are made or shut down. Problems such as dropped connections, data corruption, and password failures are logged here. Problematic connections can be diagnosed by reviewing the file, and these files can even be sent in to technical support staffs for serious problems.

InterNiche embedded PPP uses a similar logging system, however it has to take into account that many embedded systems don't have traditional file systems. For this reason, PPP logging is done by calling the routine ConPrintf(), which has the same syntax as printf().

```
ConPrintf(char char *, ...);
```

ConPrintf uses NicheStack’s GIO mechanism. Where data is sent by ConPrintf can be controlled either at compile time, or dynamically via the pdebug menu function. If PPP debugging is on, then by default the data will be sent to stdout, which is either the console or the application that issued the pdebug command (e.g., telnet). Alternatively, if PPP_LOGFILE is defined, the output will be sent to a log file if PPP_LOG_TO_FILE was set at compile time or if logging to a file was specified via the pdebug command.

Setting Line Types

InterNiche PPP customers who will only be using modems as PPP links will not need to worry about setting line type. The Hayes modem type is the default type in these situations, and nothing needs to be added by the porting engineer. However, if you will be using a combination of the supported line drivers or adding a driver of your own you will need to understand the hooks described in this section.

The first task of supporting multiple or unusual line types is assigning the type when the line is created. InterNiche PPP supports this with a callback mechanism.

`ppp_type_callback()`

The variable ppp_type_callback is actually a function pointer in the PPP code. The function indicated is called whenever a new PPP connection (mppp) with a line type of LN_PORTSET is created. If the pointer is NULL then the callback is not called and the connection creation fails.

To control the types assigned to PPP connections when they are created, the porting engineer should set the global integer `ppp_deafult_type` before any PPP connections are created (e.g. at system startup). Setting this to one of the LN_ types other than LN_PORTSET will result in every connection being assigned to that type. For systems that mix two or more types, `ppp_default_type` should be set to LN_PORTSET and the pointer ppp_type_callback should be set to a port-provided routine. When the PPP connections are created, the ppp_type_callback will be called, allowing the port's routine to assign the PPP types in any way it needs to.

For an example of how to write and use this callback routine, please obtain the reference implementation for Windows. This port supports PPP over modems, raw serial ports, and PPPoE. It demonstrates how application code can control mixed types on multiple PPP interfaces.

Adding New Types of Line Driver

To add new types of line drivers to the system there are two steps. First, the porting engineer must first provide the code for the routines described in Line Driver Calls. Second, a routine must be provided that is named ppp_portlinksetup(). The syntax for this is shown here, and the function is described at length in Porting Programmer Provided Routines.

```
int ppp_portlinksetup (M_PPP mppp);
```

This routine should be called if a line device is used other than the ones supplied by InterNiche. The porting programmer should provide code to fill in the line device table in the passed mppp with pointers to his line driver entry points. This is also a final opportunity to do any initialization required by the line driver code.

This routine is only referenced if the #define USE_PORTLINKSETUP is defined in ppp_port.h or ipport.h. This way, ports that don't need this service don't need to define the routine.

Authentication - User-Name and Password Support

Most PPP implementations will need some form of authentication - user-name and password support. InterNiche's reference ports provide a simple database of user-name and password information but if your system already has support for a user database you will probably prefer to map the PPP authentication routines into your own.

This is done by replacing the two simple routines in ppp_port.c which implement the authentication lookups. These are **get_secret()** and **check_passwd()**. Both are described in detail in Porting Programmer Provided Routines.

Timer tick

All ports must provide a timer tick so that PPP can perform timeouts and retries. This is done by making sure the system periodically calls the routine **ppp_timeisup()**, described in The Clock Tick.

Memory Allocation

All ports must provide the primitive to allocate and free the mppp structure. In systems which only support one link this may be as simple as having a single static mppp buffer and returning a pointer to it from the alloc routine (described in Dynamic Memory). The free routine can be a "no-op" (do nothing).

Configuring PPPoE Links

This section only applies if you have the PPPoE code.

PPPoE presents a collection of options that can be configured during runtime. These fall into three main areas:

- Mapping PPP connections to Ethernet devices
- Supporting PPPoE Tags
- Managing PPPoE Access concentrator and service names

See the chapter on PPP over Ethernet (PPPoE)for details of this.

IP6CP

For IP_V6 over PPP, IP6CP uses a LINK-LOCAL address that has a format of FE80::XXXX:XXXX:XXXX:XXXX, where each ‘X’ represents a hexadecimal number. In IP6CP the last 16 bytes are called the Interface ID (IFID). The IFID must be unique on the network. PPP will create the IFID using the first method in below that succeeds.

1. If an IFID is specified via a ppp config command, PPP will use that.
2. If a non-null IFID is defined in IPCP6_H, PPP will use that.
3. (Preferred) If a MAC address is available, PPP will automatically create a unique IFID based on the MAC address.
4. If a timer is available, PPP will create a random IFID based on the timer.
5. During connection negotiation for IP6CP, PPP will request that the peer create an IFID for us.

No matter which manner is used to create the address, IP6CP should negotiate the address with its peer. The peer will only reject an IFID if it is a duplicate of its own IFID. In this case, the peer will suggest another IFID.

IP6CP provides a configuration option that determines whether or not the local system is willing to accept an IFID suggested by a peer. The option can be defined in ipcp6.h (CAN_CHG_IFID) or it can be set via the ppp options menu. If this option is set to 0 (ipcp6.h) or “No” (via the menu), then if the peer rejects our specified IFID, PPP will close the connection.

4.5 Testing

Once you have included PPP in your target system, you need to test (and possibly debug) it. We recommend the following sequence of tests.

Loopback

A good first test is to set up a PPP loopback driver in crossover mode (see LB_XOVER compile option in PPP Options) and ping it. The recommended IP address of your loopback driver's interfaces is 127.0.0.1, and 127.0.0.2. The loopback driver should establish an LCP connection between the two crossover units, acting as a client on the unit which sends the ping, and as a server on the crossover unit. The ping packet should then go out one unit and in the other. Packet and byte counters at the IP interface layer should reflect this behavior. In the event this does not happen smoothly, the best approach usually is to trace the execution with a source level debugger. Since all the events take place in a single system, you can debug this basic functionality without the complexity of having to monitor two separate systems.

Client Connection

The next test should be to try a client connection via a real line driver. Send a ping to an IP host available via the IP address you have assigned to the PPP link The PPP code should call ln_connect() for the link, and when that returns successfully send a series of LCP negotiation packets via ln_putc(). LCP will not go to the connected state until it receives the correct LCP responses from the PPP host connected to. 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 still the best tool. At the point where LCP packets are being exchanged you want to turn on the logging feature (see ConPrintf()), to get a higher level look at what's happening during LCP negotiation.

Debugging LCP connections with a remote machine is probably the most complex part of most PPP ports. You should have a copy of the PPP RFC documents handy, preferably a hard copy, and be prepared to examine the logged LCP option negotiation packets in detail.

For initial testing you should simplify the negotiations by disabling CHAP, VJ compression, and DHCP. Once basic byte transfers work, the most likely source of initial LCP problems result from one side insisting on a set of options that the other side will not support. Once you can establish an LCP connection, turn these options on one at a time and re-test. Keep in mind that a connection problem may also be due to your embedded system NOT using an option that the other side insists on. Most commercial ISPs, for example, will not establish a connection unless you use an acceptable form of authentication.

Server Connection

The next test we recommend is to set your line hardware in auto-answer mode and let another PPP machine call you. Debugging this is similar to client connect debugging, but a little more complex.

Abrupt Disconnect

You will want to make sure that a broken connection will not permanently disable your PPP layer. This is usually not a problem when PPP initiates the disconnect via a TERMREQ LCP packet, but an unexpected line failure must be sure to call PPP via the `ppp_lowerdown()`call.

5 PPP over Ethernet - PPPoE

PPP over Ethernet is protocol option available with the InterNiche PPP software. It implements PPP connections over Ethernet as described in RFC2516. The PPPoE software establishes virtual Point to Point connections over the Ethernet hardware, handling Ethernet Multidrop addressing issues invisibly to the upper layers of PPP.

PPPoE is a complex protocol in its own right. In includes a discovery mechanism for finding other PPPoE nodes on the Ethernet, a negotiation phase to exchange the "TAGS" (see RFC2516) which will be used during the session, an encapsulation header for the PPP datagrams, and even some security.

When used with InterNiche TCP/IP and PPP, the PPPoE (with the default tags) should require very little special attention on the part of the porting engineer. If your target hardware has a single Ethernet interface, then all you need to do is assign an "Access Concentrator" name to your machine. If you will be using multiple Ethernet interfaces then you will also have to construct a map indicating which PPP sessions should use which Ethernet. There is also a callback mechanism that allows you to control what PPPoE TAGS are used.

5.1 PPPoE Tags

RFC2516 describes a number of tags which the PPPoE hosts use to exchange information. These tags contain user configurable information, and thus the PPPoE code needs to provide a mechanism to set default values for these tags, and also to negotiate/verify tags during runtime. This is done via callbacks to two routines in the file poe_port.c. The routine poe_setoption() is called by the PPPoE code when it wants the local "tags" to be set for a new connection, and poe_checktags() is called when a tags list is received from the peer. Its function is to check the tag list and return a code indicating if the list is acceptable. Both these functions are described in detail in PPPoE Callback Functions.

The routines in the default ppp_port.c file support the tags "Access Concentrator" and "service name". A single system-wide value is assumed, as described in the next section.

The code provides stubs for other options described in the RFC, however the code for these cases simply returns a rejection code. The porting engineer is expected to add the support for any needed options by modifying the code.

5.2 Access Concentrator and Service Names

Almost every PPPoE port will want to allow the end user to change the Access Concentrator (AC) name. The AC name is usually a text string (described in RFC2516) that uniquely identifies every PPPoE host on an Ethernet. An AC may also support a Service name (again described in RFC2516). The AC and Service names are not necessarily null-terminated "C" strings, so a length field is also required.

The service name is optional. Its use is up to the application; if not used it may be left set to NULL. This is the default for the InterNiche PPPoE as shipped. Examples of the use of the Service-Name TAG are to indicate an ISP name or a class or quality of service. Like the AC name, the service name is not required to be a C string, so a length field is also required.

The AC and Service name variables are defined in `ppp_port.h`.

For initial testing you may use the default values, or change them in the poe_port.c file before compiling. For production, you will most likely want to change the variables at run time. This should be done during system initialization - changing these values once the system is on line could confuse other hosts on the network.

Products using PPPoE should generally require the end user to configure an AC name and Service name when the product is installed, and save these items in non-volatile storage (i.e. flash). As part of system initialization the names should be read from flash and the pointers and lengths in ppp_port.c set appropriately.

5.3 PPPoE Callback Functions

poe_setoption

Name

`poe_setoption()`

Syntax

```
u_char *poe_setoption(u_short tagtype, int *len, u_char code, struct
poe_line *line)
```

Description

poe_setoption() is called by the PPPoE code to obtain the local "tags" to be set for a new connection. Before sending each packet during the creation of the new connection, the PPPoE code calls poe_setoption() once for each tag type in the array poe_tags[]. The tag type, type of packet being sent, and pointer to the connection's poe_line structure are passed as parameters.

The line structure parameter may be used to uniquely identify each client PPPoE connection. Server connections pass a NULL for the line structure since they do not actually acquire a line structure until late in the negotiation process.

If both the returned value and the len pointer are non-null; then the returned data will be copied into the tags list of the outgoing packet. A return of NULL means the tag is not supported by the port, and no data is placed in the outgoing tags field. A tag which is supported but has no data in indicated by a non-NULL return and a zero returned in the returned length. This results in a dataless entry of that tag being placed in the tags field. This is the default handling for the "Service Name" tag.

Returns

Returns NULL if tag is not supported.

Returns non-null value with `*len` set to zero for a tag which is supported but has not data associated with it. In this case the returned value does not need to point to actual data, it just needs to be non-NULL.

Returns non-null pointer to data and a non-zero length if actual data is to be used for the tag value.

poe_checktag

Name

`poe_checktag()`

Syntax

```
int poe_checktag(u_short tagtype, u_char code, u_char *data, u_short
datalen)
```

Description

This routine should examine the tag data passed and verify if the passed tag is acceptable. This single routine is used for all tag types. Optional tags like "cookie" can be implemented by editing this routine (in coordination with poe_setoption()).

This is called once for each of the non-error tag types. This means that a single received packet will generate seven calls. The reason for including tags that do not appear in the packet is so that systems that require cookies or vendorspec tags can catch their omission and treat it as an error.

The first two passed parameters indicate information about the received packet. The tagtype is one of the TAG_ defines. The code is one of the PPPoE packet type codes (e.g.: PADI_CODE). The default code illustrates how these can be used to determine if we received the packet in the role of client or server.

If the passed data pointer is set and datalen is nonzero, then data points to the tag information from a received packet and the data field is datalen bytes long. If the data pointer is NULL then the received packet contained no information for that tag type. If data is set and datalen is zero, it means the tag code was found in the received packet's tag list but contained a zero length data field (i.e. no data).

Returns

Returns 0 if the tag is OK, else -1.

6 Modem Dialer Code

Although the modem line driver code is not strictly part of PPP, it is commonly used in embedded systems which use PPP and InterNiche provides the this code along with the sources. This chapter explains how it works, and how to adapt it to new UART hardware.

6.1 Modem Code Source Files

The modem sources are shipped in the modem directory, which is a peer to the ppp directory. The sources files are:

File	Description
dialer.c	The bulk of the portable Hayes modem code.
modem_mod.c	Contains the modem task and module arrays and the modem prep, init, start, and close functions.
mdmport.c	Port dependent modem code.
mdmport.h	Port dependent modem includes.
modem.h	Hayes modem definitions.
modem_nt.c	Code for modem menu functions and the modem menu and parameter definitions used by the CLI. The porting engineer may modify or add to these.

The two mdmport files are intended to contain all code that may need to be modified when porting this code. In practice, this code can be configured for almost every port by a few #ifdefs and these files almost never need to be modified.

Like all InterNiche source directories, the modem directory includes a Microsoft nmake compatible makefile that compiles the modem code into a library. Some other build systems are supported too. Contact InterNiche support for the latest list for supported code building systems.

6.2 Modem Line Driver

To the PPP code, the modem code looks just like any other line driver (see Line Drivers). It is implemented as a series of C routines that are organized into a line structure. The dialer also makes lowerup and lowerdown calls to the PPP code when the modem line is connected or disconnected.

The use of multiple simultaneous modems is supported. When the PPP link is created for a modem interface, the code in ppp_line_init() in pppsys.c assigns the line driver (in this case the modem driver) entry points to the line structure embedded inside the mppp structure. It then calls modem_init(), passing the address of the line structure inside the mppp structure. The modem code uses the mppp line structure address (which is passed to the modem calls) as an identifier so that output can be sent to the correct modem when multiple modems are in use.

It seems a reasonable assumption that the design of a system will know how many modems can be attached to it, so the number of modems which can be supported is given by the #define NUM_MODEMS. A static array is created which contains a structure struct atmodem to manage each modem. This structure contains the pointer to the upper layer (PPP) line structure, a unit number for the lower layer (UART), and state information about the modem itself.

6.3 Non-Volatile Modem Parameters

The modem code requires two data parameters: the modem initialization string and the telephone number. These are stored as C strings in the character arrays `mdm_init_string[ ]` and `mdm_dial_string[ ]` respectively. These parameters must either be entered via script commands each time the system boots or saved across boots by some method defined by the porting engineer. As provided, InterNiche's PPP supports a single initialization string and telephone number. If multiple strings or numbers are required, the porting engineer will have to add this support to the file `dialer.c`. Perhaps the simplest way of doing this is by replacing the character arrays described here with calls to C routines. They can pass the modem units or PPP line pointers to identify which connection is being made and return pointers to strings which can be used on a per-modem basis.

6.4 Modem Compile-Time Options

The modem code supports several compile time options. These should be #defined (or not) within the scope of the ppp_port.h file. These options are:

USE_MODEM

This option is required to compile and use the modem code. It is provided so that inflexible build systems (i. e. many GUI based "project" files) can support compilation with or without the modem code

MDM_CHECK_NO_CARRIER

This includes code to scan the UARTs input stream for the pattern NO CARRIER. The modem will send this test to us via the UART if it detects the it has lost the carrier signal (i.e. we have been disconnected). Unfortunately, this pattern could conceivably appear inside legitimate PPP data being sent to us over the modem connection. To avoid mistaking this data for an actual loss of carrier on the modem, the detection code also uses a timing mechanism. The NO CARRIER message must be preceded by a short idle period, or we assume it is data and not a modem message.

This option should be used only on systems that cannot use the MDM_DCDLINE option described below.

MDM_DTRRESET

Like all devices, the modem needs to be reset from time to time. The most reliably way to do this is to drop the DTR line for 0.5 seconds and then raise it again. This #define includes code to allow us to use this reset mechanism. Unfortunately many serial cables, and even some bad designed boards, neglect to wire the DTR line; so the less robust AT command method of transmitting a "+++" sequence is also used. If your system will have a working DTR line, however you should use it by setting the define.

Note that is you set this define you will need to add two simple routines in your UART driver - modem_clr_dtr(); and modem_set_dtr(). The first simply lowers the UART DTR line, and the second raises it. These are one op-code instructions on most systems.

MDM_DCDLINE

This #define includes code which uses the RS-232 "DCD" (Data Carrier Detect) line to detect when the modem has lost the carrier (i.e. the connection has been lost). This is a more reliable method of detecting loss of carrier than the software enabled by MDM_CHECK_NO_CARRIER. Unfortunately, like the DTR line, many target systems are deficient in that the UART connector hardware is missing this line. If is true in your design, make sure that MDM_DCDLINE is not enabled and that MDM_CHECK_NO_CARRIER is enabled.

6.5 UART Driver API

This section describes the UART routines that the modem code uses to communicate with the modem. They are simple routines that should be easy to implement on any existing UART driver.

Modem Unit Numbers

In order to support multiple modems, each of these calls takes a "unit" number as a parameter. Each modem support structure uses a unique unit number in the range of 0 -to (NUM_MODEMS - 1). Systems supporting one modem unit (i.e. NUM_MODEMS == 1) can ignore the unit number. Multi-modem systems will need to provide a mechanism to map the unit numbers to the UART devices.

The UART driver calls are designed to take advantage of UARTs that can buffer characters. The send call uses semantics which allow it to post characters to the UART at higher speeds than the UART can transmit them to the modem. If the UART cannot accept characters being posted to it, the sending task will block briefly. If the UART driver then sends all posted characters before the task resumes and tries to send some more, then the UART will be idle until the blocked task can resume. Given the typically slow performance of UARTs (relative to other network media) this should be avoided. The PPP code will run more efficiently if the UART can buffer a maximum sized packet's worth of characters - usually about 1520 bytes.

Similarly, the receiving routine is designed to be non-blocking. The UART receive routine is typically called by a single timer driven thread. Since this thread also supports many other timer functions throughout the InterNiche system, it must not block waiting for UART input. Instead, it returns to its other duties, and then sleeps until the next time tick wakes it up. Once awake, the thread will collect and process all the bytes the UART has buffered while the thread was sleeping. Since the thread may sleep for up to half a second, a UART running at 56Kbaud could potentially receive up to 3.5 K bytes (56K / 8 / 2). On typical system the timer will run at least 20-30 times a second, which still requires a buffer of about 350 bytes.

Another issue to consider is character loss. UARTs with small (or nonexistent) data FIFOs and no hardware flow control are highly likely to drop received characters. A 56K baud UART may receive 7000 characters a second, or one every seventh of a millisecond. If the system is unable to service the UART interrupts for that long, then characters will be lost. Remember that PPP checksums its packets, so a single missing byte causes an entire packet to be dropped. In a situation like the one just described, it is quite likely that no PPP packets will ever be received intact.

This situation can be detected by the presence of large numbers of "FCS" errors in the PPP log, which is described in Reading Log Files. This may happen even though the UART had already passed some early unit testing. The tests are often conducted without other (non-UART) interrupts occurring in the system, or with short packets, which are much more likely to be received intact than long packets.

The solutions for this are obvious, although not always easy. Some suggestions are given below. The more of these you can implement, the better.

- Ensure the UART supports a large internal received data FIFO
- Provide hardware flow-control handshaking
- Reduce the system's interrupt latency to a minimum

While most of the sample InterNiche UART drivers use interrupts, this is not required. A "polled mode" UART driver may be used, however there should be a FIFO or DMA buffer large enough for at least 1500 bytes of data.

In the course of writing these routines for a new UART, it is quite useful to have example routines. InterNiche can provide these for a variety of popular embedded systems UARTs. There is also in implementation available for Microsoft Windows "Comm" port API that supports up to two modems on the Windows devices Com1 and Com2. If you don't have a working example of a UART device for reference, contact InterNiche to obtain one.

uart_init

Name

`uart_init()`

Syntax

```
int uart_init(int unit);
```

Description

This is called from within the modem line ln_init() call. It should prepare the UART for IO to the modem. This can include initializing the hardware and installing required ISR. When this routine returns 0 the next call from the modem code will most likely be sending characters.

This routine will be called once each time the modem is to be connected by the PPP code. On most systems, initializing the UART hardware and installing ISRs should only be done on the first call. Subsequent calls may simply return a 0 (SUCCESS) if the UART for the passed unit remains ready for use.

Returns

Returns 0 if OK, else returns one of the ENP_ error codes.

uart_getc

Name

`uart_getc()`

Syntax

```
int uart_getc(int unit);
```

Description

This call should return the next received character that is ready at the UART driver in the low order 8 bytes of the returned value. The upper bits of the returned value should be zeros.

If no character is ready, a -1 (all bits set to 1) should be returned. This routine must not block waiting for new data. See the discussion in the previous section.

Returns

Returns 0 if OK, else returns one of the ENP_ error codes.

uart_putc

Name

`uart_putc()`

Syntax

```
int uart_putc(int unit, u_char char);
```

Description

Send a character out the UART. The character to send is passed in the low order 8 bits of the char parameter.

Returns

Returns 0 if OK, else returns one of the ENP_ error codes.

uart_stats

Name

`uart_stats()`

Syntax

```
int uart_stats(GIO * gio, int unit);
```

Description

Display UART statistics to the output device gio passed. The device is simply passed as a parameter to the InterNiche console routine ns_printf().

Returns

Returns 0 if OK, else returns one of the ENP_ error codes.

uart_ready

Name

`uart_ready()`

Syntax

```
int uart_ready(int unit);
```

Description

This is used by the modem code to find out if UART is ready to send a character. If the UART is prepared to accept a character for transmission then this routine should return TRUE, otherwise it should return FALSE.

The UART does not have to be able to actually send the character immediately, it only needs to be able to accept the character for sending. This means a UART with a send buffer should return TRUE if it has any space available in the buffer. The key point is that when uart_ready() returns TRUE, the next character passed to uart_send() must not be discarded due to a full buffer or FIFO.

An alternative implementation is to always return TRUE from this routine, and then have uart_send() block the calling thread if the UART is busy. This will work for the InterNiche modem and PPP code; however, blocking the thread which calls uart_send() may hurt system performance.

Returns

Returns TRUE if the UART is prepared to accept a character for transmission, otherwise returns FALSE.

7 Reading Log Files

The debugging text output by the calls to ConPrintf() can quite useful to the developer during development of the PPP product as well as being vital to the end user once the product has shipped. This chapter provides some guidance to reading and interpreting these logs.

7.1 What Gets Logged

Log files can be files directly created on a local file system by ConPrintf (), or they may be captures of output made to a console. An example of this latter case might come from a system that has no local file system, but sends console output to a free UART. The UART would be hooked to a terminal emulator such as windows HyperTerm. The text that ConPrintf () sends to the UART will appear in the HyperTerm window, where it can be selected and copied to a file. Throughout this section, both types of record are referred to as "log files".

If `PPP_LOGFILE` is defined and logging is enabled then all the output from ConPrintf() goes into the log files, and ONLY output from ConPrintf() ends up in the log files. This means that if you want to find the source of a message in the log file, all you have to do is grep for the ConPrintf statements in the source code. Somewhere there will be a ConPrintf() statement that produced the line you are interested in. Keep in mind the output lines are produced from printf-like format strings, so output like "`length = 7`" might have come from the string "length = %d".

Log files are more than simply error logs. They primarily contain progress reports and event notifications as well as error reports. They may also include events from software modules that are related to PPP but not strictly part of it. Examples of this are the modem dialer and UART drivers.

If you are using a modem, the log text from the modem dialer will probably occupy a number of lines at the beginning of every connection. These include copies of each command sent to the modem and each reply received in response. This allows you to:

- Verify the phone number you dialed
- Detect busy signals and other non-answering conditions
- Spot timeouts when the modems can't "Train"
- Learn the baud rate at which you connected
- Spot failures where the modem simply lost the connection
- Detect that the modem is powered off or not connected to the UART
- Detect hookup cables that don't support the DCD line

The process of connecting and configuring a modem with an embedded system is very error prone, and the log file is a vital tool to help both the developer and the end user get through this successfully.

The bulk of the log contents from the PPP code itself will usually be notifications of option negotiation. As explained in RFC1661, each PPP layer contains a Finite State Machine (FSM), where the "negotiation" of configuration options is a primary source of state transition events. If the two PPP hosts cannot negotiate a mutually acceptable option set the PPP connection will fail. To correct this situation, it is vital for the end user (or his technical support people) to know just what options were tried and rejected.

7.2 Option Packets

Each PPP protocol has a specific set of options, which are described in detail in the protocols RFC document. The main protocols that concern InterNiche PPP users are LCP, IPCP, IP6CP and authentication (either PAP or CHAP). The two peers on the PPP line negotiate by exchanging packets containing encoded lists of options. The PPP header of each of these packets has a one-byte code indicating that it is one of the following types:

RFC1661 Name	Mnemonic in Log File	Description
Configuration-Request	CONFREQ	A requested option list
Configure-Ack	CONFACK	The options are OK
Configure-Nak	CONFNAK	The options are not OK but an alternative value is suggested in the echoed list.
Configure-Reject	CONFREJ	Option(s) not OK and no alternative is given

As the protocols exchange these option packets, the code and option list contents are recorded in the log. Since we use nearly every option described in the RFCs they are not listed again in this document.

It is useful to understand the order in which the protocol negotiations occur, since this will be reflected in the log file. This is generally:

1. Line device (i.e. Modem dialing).
2. LCP.
3. Authentication.
4. IPCP or IP6CP.

As mentioned above, the first entries in the log will those involved in the line device (i.e. the modem) connection. A line device is becoming ready for PPP use (i.e. the modems have connected) is known as a lowerup event. PPP is notified of this event through a call to **ppp_lowerrup()**. The line device passes ppp_lowerrup() a protocol code of LCP_STATE, since this is a lowerup event for the LCP protocol.

When the lowerup event occurs, the upper layer (in this case LCP) immediately sends a Configure-Request packet, and of course the contents of this packet are logged in the log file. Here is an example of an LCP Configure-Request packet from an actual log file. Some linefeeds have been added for clarity.

```
ppp_inchar: got PPP_FLAG ppp_inchar: got PPP_FLAG ppp link 00BC8328 rcvd:
C0 21 01 02 00 1C 01 04 05 DC 02 06 00 0A 00 00
03 04 C0 23 05 06 7C 79 AD 9D 07 02 08 02
ppp_inpkt (link 00BC8328); prot: LCP, code CONFREQ, state REQSENT, len 24
lcp_reqci: rcvd MRU[2176] (1500)(ACK)
lcp_reqci: rcvd ASYNCMAP[2176] (000A0000)(ACK)
lcp_reqci: rcvd AUTHTYPE[2176] (C023)(ACK)
lcp_reqci: rcvd MAGICNUMBER[2176] (7C79AD9D)(ACK)
lcp_reqci: rcvd PCOMPRESSION[2176] (ACK)
lcp_reqci: rcvd ACCOMPRESSION[2176] (ACK)
lcp_reqci: returning CONFACK.
```

Many of the ConPrintf() strings start with the containing C function name followed with a colon. Thus we can see that the first line was output from the routine ppp_inchar(). This line is recording that PPP received the single byte PPP flag character that delimits PPP packets. If all goes well, a complete PPP packet should follow.

The next line is a hexadecimal dump of the received PPP packet. We will analyze this some more below. By default, the entire packet is recorded as a hexdump to the log. The number of bytes in the hex dump may be limited by using the appropriate option of the `ppp pdebug` menu command. Setting `PPP_HEXMAX` to zero in `ipport.h` will prevent all hex dumping. This can be useful when logging is being done to a device which is slow or has limited memory.

The hex dump is followed by a line from ppp_inpkt(), indicating that we have received a good PPP packet. If the packet had contained an FCS error (data corruption) this would have been noted here instead.

7.3 Reading the Hexadecimal Packet Captures

Returning to the hex dump, the first six bytes are the PPP type field and header. It is useful to understand how these are parsed:

```
C0 21 - protocol type. C021 is LCP
01    - packet type. 01 is Configuration-Request
02    - Packet Id, for detecting retries.
00 1C - length of packet, not including the protocol field
```

The rest of this log sample is form the routine lcp_reqci(), detailing the contents of the options. Each option is a type (specific to the protocol, in this case LCP), a length, and possibly some parameter data for the option. Let's parse the first option as an example:

```
01    - LCP "MRU" (Max Receive Unit) option
04    - length of option, including type and length bytes
05 DC - option parameter, in this case MRU of 1500.
```

All the various LCP options, including their assigned values and formats, are described in RFC 1661.

7.4 Authentication

One of the LCP configuration options is to select an authentication protocol. This is not mandatory, however it is used on most systems. Even if your PPP product does not use authentication, you will find that most ISP servers require you to use CHAP or PAP to log in before they will let you complete the IPCP layer connection. Because of this, it is a good idea to know how to spot the authentication protocol negotiation in the log.

In the LCP packet example above, the AUTHTYPE option indicates the authentication protocol being requested. Here's how the AUTHTYPE option field parses:

```
03    - LCP AUTHTYPE option
04    - length of option field
C0 23 - Authentication protocol desired.
```

The AUTHTYPE parameter is C023, is the defined value for the PAP authentication protocol. The other likely alternative in this field is C223, which is the code for CHAP. Here's a CHAP option parse:

```
03 - LCP AUTHTYPE option 05 - length of option field C2 23 - Authentication proto

        CHAP "digest type" parameter
```

The CHAP option is one byte longer, since it includes an extra parameter to indicate what digest type to use. Currently the only correct value for this is 05, which indicates the RSA MD5 message digest algorithm. If the machine at the other end of the connection sends a digest type of 80 or 81 (hex) it is attempting to use MS-CHAP or MS-CHAPv2 respectively. (See User/Password Authentication Protocol).

Once the LCP negotiations complete successfully, the indicated Authentication protocol will be started. Authentication protocols do not use the standard PPP FSM, so this event is not strictly a lowerup, however it is somewhat analogous. Depending on the authentication protocol agreed upon, this may just require the machine to prepare for a packet form the peer. In other situations it means sending the first authentication packet.

The result of the authentication transaction will always be logged. If authentication has failed, this will appear near the end of the log. The most common cause of this is an incorrectly typed user-name or password, so double-check this before you call InterNiche support.

If authentication succeeds, a lowerup event will be sent to the IPCP layer, and IPCP option negotiations (similar to the LCP options described above) will commence. IPCP has fewer options than LCP, and the only one is likely to cause a negotiation failure is the assigning of the IP address. This happens most often when the client machine wants to obtain an IP address, and the server cannot assign one. If this happens it will be readily apparent from the log. You will need to reconfigure the server to resolve the problem, or obtain a useable IP address some other way.

8 PPP Menu Options

8.1 ppp config

Command Name

`ppp config - Configure or display PPP global parameters.`

Syntax

```
ppp config  [-a <AC name>] [-e <secs>] [-t <secs>] [-z <Service name>]
[-c <req | pref | no>] [-m <md5 | mschap>] [-p <req | pref | no>] [-s <CHAP sec

[-d <req | prov | no>] [-x <ipaddr primary>] [-y <ipaddr secondary>]
```

Parameters

`-c`	string: Require CHAP, `"req"` or `"pref"` or `"no"`
`-m`	string: Set preferred CHAP type, `"md5"` or `"mschap"`
`-p`	string: Require PAP, `"req"` or `"pref"` or `"no"`
`-s`	string: Secret for CHAP authentication
`-a`	string: Access Concentrator name tag
`-e`	Interval in seconds between echo requests. 0 = No echo requests
`-t`	Line timeout in seconds. 0 = No timeout.
`-z`	string: Server service name tag
`-d`	string: DNS server addresses, `"req"` (request) or `"prov"` (provide) or "no"
`-x`	IPv4 address to provide to peers as their primary DNS server
`-y`	IPv4 address to provide to peers as their secondary DNS server

Description

This command configures PPP global parameters. The options 'c', 'm' 'p' and 's' are related to CHAP and PAP. Options 'a', 'e', 't' and 'z' are specific to PPPoE. Command options 'd', 'x' and 'y' pertain to setting PPP DNS server information.

Notes/Status

- ppp config without any options displays the current state of the dynamically configurable PPP global variables.
- Options `'-c'` and `'-s'` are only available if CHAP_SUPPORT is defined.
- Option `'-p'` is only available if PAP_SUPPORT is defined.
- Option `'-m'` is only available if both CHAP_SUPPORT and MSCHAP_SUPPORT are defined. The value set will be requested, but negotiation may result in the other type being used.
- For Options `'-c'` and `'-p'`, "req" means negotiations will fail if the peer does not agree to the required protocol. "pref" means, if the local system is the client, it requests this authorization protocol. If the local system is the server, it NAKs the first request for a different authorization protocol. However, if the peer persists, it accepts the alternate protocol if available. "No" means not required. The local system freely negotiates any of the available options.
- Options `'-a'`, `'-e'`, `'-t'`, and `'-z'` are only available if USE_PPPOE is defined.
- Option `'-t'`, the idle timeout, must be larger than option `'-e'`, the echo request interval.
- Option `'-d'`, `'-x'`, and `'-y'` are only available if PPP_DNS is defined.
- If option `'-d'` is set to "req", the local system will request DNS server addresses from the peer. The current values (may be zero) in the dns_servers[] array will be provided in the request. The peer may accept these or provide new DNS server addresses. New addresses will be added to the end of the array, if there is room. Otherwise, they will be ignored.
  Note: If there are no addresses in the local dns_servers[] array and local system requests DNS server address and the peer does not provide at least one, then IPCP negotiations will fail.
  If option `'-d'` is set to "prov", the local system will provide DNS server addresses to any peer that requests them. The parmary and secondary DNS server addresses that will be provided can be set either in ppp_port.h or by the `'-x'` and `'-y'` options to this command. At least one address must be non-zero. A zero address will not be provided to the peer
  If option `'-d'` is set to "no", PPP will no longer request or provide DNS server addresses
- For options `'-x'`, `'-y'`, if the addresses already exist, they are replaced by the new values. Note: these values have no effect on the values in the local dns_servers[] array.

Location

This command is provided by the ppp module when USE_PPP is defined.

8.2 ppp netstat

Command Name

`ppp netstat - Display PPP status and statistics.`

Syntax

```
ppp netstat [-l] [-n < iface name>] [-p] [-s]
```

Parameters

`-l`	(default) Displays a brief summary of the status of all of the PPP links in the system.
`-n`	string: "ALL" or interface name in the form shown by the iface command (e.g., "pp0").
`-p`	Displays a summary of PPPOE links.
`-s`	Displays PPPOE sessions.

Description

When used without options or with the `-l` option, this command displays a brief summary of the status of all of the PPP links in the system. There is one line of output per link.

Sample output:

```
index  link_addr   iface  flags   type    LCP      IPCP
   0   00BC8328     pp0   80    ATMODEM INITIAL  INITIAL
```

When used with `-n` it displays extensive information about the specified link(s). When used with `-p` it shows configuration info. for all pppoe links, 2 lines of output per link.

When used with `-s`, it displays a summary of the status of active pppoe sessions, 3 lines of output per session.

Sample output:

```
   state:      retrys:      ID:      iface:
   Service:        AC:
   pkts; in:      out:     last-ctrl:       last-rxdata:
```

Notes/Status

- The `-p` and `-s` options are only available when USE_PPPOE is defined

Location

This command is provided by the PPP module when USE_PPP is defined

8.3 ppp pdebug

Command Name

`ppp pdebug - Configure PPP debug options`

Syntax

```
pdebug [-d <on | off>] [-f <on | off>] [-h <on | off>] [-l <length>]
```

Parameters

	"pdebug" without parameters displays the current state of debug options
`-d`	string: Turn on debugging, `"on"` or `"off"`
`-f`	string: Send debug messages to PPPLOGFILE, `"on"` or `"off"`
`-h`	string: Hexdump PPP packet data, `"on"` or `"off"`
`-l`	integer specifying the maximum length of the hexdump for a message.

Description

This command configures PPP debug options

Notes/Status

- If `-f` is set, data will be written only to PPPLOGFILE.
- One of the options `-d` or `-f` must be set in order to turn on hexdump.
- Option `-f` is only available when PPP_LOGFILE is defined.
- Options `-h` and `-l` are only available when PPP_HEXDUMP is defined.

Location

This command is provided by the `ppp` module when `USE_PPP` is defined.

8.4 ppp plink

Command Name

`ppp plink - Control a PPP interface`

Syntax

`ppp plink -n <iface name> -d | -u`

Parameters

`-n`	string: interface name in the form shown by the iface command (e.g., "pp0")
`-d`	Take the specified interface down
`-u`	Bring the specified interface up

Description

This command controls a PPP interface

Notes/Status

- Option `'-n'` is required. Only a single interface name can be entered.

Location

This command is provided by the `ppp` module when `USE_PPP` is defined.

8.5 ppp plnkcfg

Name

`ppp plnkcfg - Configure PPP per-interface parameters`

Syntax

```
ppp plnkcfg -n <iface name> [-a <yes | no>] [-c <yes | no>]
    [-d <yes | no>] [-i <IPV6CP IFID>] [-j <yes | no>]
    [-m <mtu>] [-p <password>] [-r <mru>] [-u <username>]
    [-v <ip4 | ip6>]
```

Parameters

`-n`	string: "ALL" or interface name in the form shown by the iface command (e.g., "pp0"). The options below will apply to this interface or all PPP interfaces
`-a`	string: Allow peer to set our local address, `"yes"` or `"no"`
`-c`	string: Can negotiate local link's IPV6IFID, `"yes"` or `"no"`
`-d`	string: Use DHCP, `"yes"` or `"no"`
`-i`	string in the form XXXX:XXXX:XXXX:XXXX that specifies an IP_V6 IFID to use, where each 'X' represents a hexidecimal digit. This IFID will be used for the link specified by the `-n` option
`-j`	string: Use VJ Compression, `"yes"` or `"no"`
`-m`	integer specifying the maximum transmission unit
`-p`	string specifying the password to be sent for authentication
`-r`	integer specifying the maximum receive unit
`-u`	string specifying a username to be sent for authentication
`-v`	string `"ip4"` or `"ip6"`. The local interface will request this version with its initial configuration request

Description

This command configures PPP per-interface parameters

Notes/Status

.

- Option `'-n'` is required. Only "all" or a single interface name can be entered. The "all" string only applies to PPP interfaces. If "all" is specified, then the options specified will apply to each existing PPP interface and to those created dynamically at a later time.
- If option `'-a'` is set to "yes", the IP-address option will be set in the IPCP configuration request. The current address, which may be zero, for that PPP interface will be included in the option. If the peer NAKs the value we sent and sends a different address, we will accept and use it
- Options `'-c'` and `'-i'` are only available if IP_V6 is defined.
- Note: "all" cannot be used with the `'-i'` option.
- Option `'-d'` is only available if PPP_DHCP_CLIENT is defined.
- Option `'-j'` is only available if PPP_VJC is defined.
- Option `'-v'` is only available if both IP_V4 and IP_V6 are defined. By default, IPv4 will be used unless the remote system specifies IPv6. The version specified by this option will be requested in the initial configuration request; however, IP4 may still be selected as a result of negotiations with the peer.

Location

This command is provided by the `ppp` module when `USE_PPP` is defined.

8.6 user

Command Name

`user - Access or modify user table`

Syntax

```
user [-a | -c] -u U [-p P] [-m M] [-r] [-z Z]
user -d -u U
user -l
user -s
```

Parameters

`-a`	Add entry in user table
`-c`	Change existing entry in user table
`-d`	Delete entry in user table
`-l`	List (dump) entire user table
`-m`	Add (OR-in) appcode bit(s) specified by string `M` to entry for specified user.
`-p P`	Set password `P` in entry for specified user
`-r`	For specified user, replace (vs. OR-in) appcode and permission fields with: `m M z Z`
`-s S`	Parameter `S` is one of the strings "TRUE" or "FALSE". It could be used to toggle whether or not the user table will be saved in file system. Implementation has been left to the porting engineer
`-u`	username `U` for -a -c or -d commands
`-z Z`	Add (OR-in) permission bit(s) `Z` to entry for specified user.

Description

Used to add or modify entries in the user table. At initialization, entries in the user table can be added via commands in a script file. No module should directly access or modify the user table. Modules should use the "user" command or call the available (non-STATIC) functions in userpass.c or user_nt.c

Notes/Status

- The `-l` option is only available if NPDEBUG is defined.
- The `M` argument is the string "all" or a comma separated list of modules (applications) for which the user entry is valid. For example, if the entry for username "root" only has the FTP bit set, then "root" is not a valid user name for any other module. Currently the individual module names in the list of modules must be one of the following: ftp, telnet, http, or ppp
- The only currently defined permissions are PERMISSIONS_ALL (0xFFFFFFFF). This field is available for the porting engineer.
- An error will be returned for the change (-c) command if the entry does not exist. With the add (-a) command, an error will be returned if the entry already exists, unless password parameter exactly matches the password entry in the table. In that case, the specified module and permission bits will be ORed-in to the existing entry.
- A username can only have one entry in the user table.
- With the change (-c) command, the values for the M and Z arguments will be ORed into the existing values of the fields in the user table, unless -r is specified. In that case the values for the M and Z arguments will replace the values in the existing entry.
- Only the username argument should be given for the delete (-d) command.
- No other arguments should be given with the list (-l) command.

9 Function Calls

9.1 Line Driver Calls

Each type of line device supported by the PPP code will require a "line" driver implementation. Line drivers are provided for Hayes modems, several UARTs, PPPoE, and a loopback test device. If you want to use any other device, you will need to create your own driver and attach it to the PPP system.

A line driver consists of four predefined C routines and support for maintaining several state variables. All the routines and variables are defined in the com_line structure, shown here:

```
struct com_line
{
   /* bring/check line up */
   int   (*ln_connect)(struct com_line * lineptr);

   /* disconnect the line */
   int   (*ln_disconnect)(struct com_line *);

   /* one of the send routines (the next two) may be NULL */
   int   (*ln_putc)(struct com_line *, int byte); /* send single char */
   int   (*ln_write)(struct com_line *, PACKET pkt);

   /* speed and state of the lower module */
   long        ln_speed; /* most recent detected speed */
   ln_states   ln_state;

   int   (*ln_getc)(struct com_line *, int byte); /* receive single char */

   /* types for the layers above and below this interface  */
   int   upper_type;
   int   lower_type;

   void * upper_unit; /* depends on upper_type, usually M_PPP */
   int   lower_unit;  /* legacy ID for lower (UART level) drivers */
};
```

Each M_PPP structure contains one of these com_line structures internally. The pointers to the driver routine are set up when the M_PPP is created (see ppp_line_init()), and a pointer to the M_PPP contained com_line is passed to all subsequent calls to line devices. The line device is expected to maintain the ln_speed member in the event its speed changes (e.g. a modem connecting at different baud rates, or a 10 /100 Ethernet). When the PPP needs to access one of the drivers line functions, it does so by calling the routines in the table.

The porting programmer must provide the routines defined and set pointers to them in ppp_portlinksetup(). All these routines may block while they do their job, although generally the only one which blocks for more than a fraction of a second is the ln_connect() call. PPP will not re-enter the routines or assume any sort of time-out.

If multiple units have the same type of hardware then the same routines can be used in all the M_PPP structures, however the line routines must be coded to use the mppp parameter passed to access the correct hardware device.

Devices may be character (byte) oriented or block oriented (packets). The code to handle both methods is #ifdef-ed in the PPP code with the PPP_CHARIO used to enable the character code and PPP_PACKETS enabling the packet handling code.

Character oriented devices, such as UARTs and modems, should deliver received characters to PPP by passing them to the line device's ln_getc() function. The function pointer is pointer is set by the PPP code in ppp_line_init(), and usually points to `ppp_direct_in()`. The reason for making the call indirectly through ln_getc() is to allow the com_line structure (and thus the modem code) to be used with line types other than PPP, such as SLIP.

Input from the packet oriented device such as Ethernet should be handled by passing received packets to a function named ppp_inpkt(). Calling this function with received bytes is considered part of implementing a packet-oriented line driver although it does not appear in the table.

The routines are defined below.

ln_connect

Name

`ln_connect()`

Syntax

```
int (*ln_connect)(int unit);
```

Description

This call will check to see if the line is connected, and initiate a connection if not. It will generally block while the connection is established. This could take a minute or more while a modem line driver dials, awaits an answer, trains, etc.

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

Returns

`0`	line is/was connected
`1`	not connected, temporary problem (line busy, etc.)
`2`	not connected, hard error

ln_disconnect

Name

`ln_disconnect()`

Syntax

```
int (*ln_disconnect)(struct com_line * linep);
```

Description

When this is called line drivers should disconnect the line. On modems, this is a hang-up. On return, the line device should be ready to initiate another connection via ln_connect().

Returns

Returns 0 if hardware hang-up event had no errors, else a non-zero error code. This return is strictly informational; PPP does not take any action based on it.

ln_getc

Name

`ln_getc ()`

Syntax

```
int ln_getc(struct com_line * line, int rx_char)
```

Description

This serves as an input function for character line drivers. It is generally set to point to `ppp_direct_in()`, which is responsible for HDLC like decoding, handles PPP input flags, and reassembles the PPP packet. Data for all protocols (LCP, IPCP, and IP should be delivered to PPP via this routine on UART-like devices.

Returns

Returns 0 if OK, else negative (ENP_) error code.

ln_putc

Name

`ln_putc()`

Syntax

```
int (*ln_putc)(struct com_line *, int byte);
```

Description

Sends a byte on the line. If the event line hardware is temporarily blocked, e.g. full FIFO, or XOFF state, the line driver should either block or queue the byte for later transmission.

Returns

Returns 0 if byte was sent without error, else a non-zero error code. If a non-zero error code is returned, PPP will assume the link has failed, dump the packet, and not retry. Indeterminate conditions, such as queuing a byte in a FIFO for sending, should return 0 unless a clear device failure is detected.

ln_write

Name

`ln_write()`

Syntax

```
int (*ln_write)(struct com_line *, PACKET pkt);
```

Description

Send a data packet on the line. The PACKET structure is defined by the InterNiche IP stack. If this mechanism is used the PACKET members will be set according to the guidelines in the InterNiche stack technical reference. Specifically, the following member variables will be set:

`pkt->nb_pro`	points to data to send (PPP header)
`pkt->nb_plen`	length of data at nb_prot

The code in the PPPoE may be used as sample code for implementing this routine.

Returns

Return values are the same as ln_putc().

9.2 Porting Programmer Provided Routines

get_secret

Name

`get_secret()`

Syntax

```
int
get_secret(M_PPP mppp,  /* IN - PPP link which is authenticating */
     char * resp_name,  /* IN */
     char * rhostname,  /* IN */
     char * out_buffer, /* OUT - buffer to put chap secret in */
     int * out_buflen,  /* OUT - length of secret */
     int flags);
```

Description

This needs to be provided by the porting programmer for systems supporting CHAP. It gets the CHAP secret stored in NVRAM and makes it available to the PPP CHAP internals. secret is copied to a buffer passed by caller, and length of valid chars is put in passed int. This is only required if CHAP is used.

Returns

Returns TRUE if OK, FALSE if problems extracting or copying secret.

ppp_portlinksetup

Name

`ppp_portlinksetup()`

Syntax

```
int ppp_portlinksetup(M_PPP mppp)
```

Description

This routine may be provided by the porting engineer to extend the list of supported line types.

It is not required, and indeed will only be called if the #define USE_PORTLINKSETUP is set in the scope of ppp_port.h.

If this is used, the PPP line type (mppp->line.lower_type) must be set to any of the LN_ types supported by ppp_line_init() in pppsys.c.This can be accomplished by setting the global integer ppp_type to the desired type, or by setting ppp_type to LN_PORTSET and providing the routine ppp_type_callback() to set the type when the line is created.

If a ppp_portlinksetup() routine is used, The porting engineer must make sure it sets the line driver routine pointers as described in Line Drivers. It should also do any per-device initialization required before returning.

This routine is also a convenient mechanism to alter other M_PPP structure variables. Items which may be modified include:

`default_ip`	default IP address.
`lcp_wantoptions`	default LCP options, such as authentication type.
`ipcp_wantoptions`	default IPCP options, such as DNS addresses

Of these, default_ip deserves a bit of explanation. IPCP may optionally set your IP address for you, and it may then be overwritten by DHCP. But in the case where you may not be getting IP address via IPCP, and will not be using DHCP, the default IP address will be the operational IP address of your IP stack on this interface. Since neither IPCP assignment nor DHCP service is universally available, it is usually a good idea to request the end user to assign an IP address (stored in NV storage) as a fallback. This can be zeros (0.0.0.0) if a DHCP assignment is required at the end user's site.

If PPP_DNS is defined, these can also include:

`dnsaddr_pri`	default primary domain nameserver IP address
`dnsaddr_sec`	default secondary domain nameserver IP address.
`neg_dnsaddr_pri`	specify if primary domain nameserver IP address should be negotiated.
`neg_dnsaddr_sec`	specify if secondary domain nameserver address should be negotiated.
`accept_dnsaddrs`	specify if peer's DNS addresses are to be accepted.

These require some explanation. IPCP may optionally get one or two domain nameserver addresses from the PPP peer for local use, and it may optionally pass one or two domain nameserver addresses to the PPP peer for its use. neg_dnsaddr_pri and neg_dnsaddr_sec are flags that, if set, indicate that IPCP should try to get primary and secondary nameserver addresses from the peer. dnsaddr_pri and dns_sec are the default values for those primary and secondary nameserver addresses, and may be zero (0.0.0.0) if there are no default values.

To receive DNS addresses from a PPP peer:

- set the neg_dnsaddr_pri flag and clear dnsaddr_pri;
- optionally, set the neg_dnsaddr_sec flag and clear dnsaddr_sec;
- set the accept_dnsaddrs flag.

Similarly, to set the PPP peer's DNS addresses:

- clear the neg_dnsaddr_pri and neg_dnsaddr_sec flags;
- set dnsaddr_pri to the DNS address being provided.

The other structure members generally do not need to be changed manually.

Returns

Returns 0 if success, or a negative error code. The ENP_ codes from the InterNiche IP stack are recommended, however any non-zero value will work.

9.3 PPP Entry Points

ppp_inpkt

Name

`ppp_inpkt()`

Syntax

```
int ppp_inpkt(M_PPP mppp, PACKET pkt)
```

Description

This should be called from the line drivers whenever a complete packet is received by the device. This is the method for packet-orient line devices such as PPPoE to pass received data to the PPP code. Character oriented devices should use the lines ln_getc() function

*The PACKET structure is defined in the InterNiche IP stack file netbuf.h. The PACKETS members should be set up as follows:

`pkt->nb_prot`	points to the PPP header
`pkt->nb_plen`	is the length of the data at nb_prot

The data in the packet should be "unescaped" and stripped of any HDLC-like stuffed bytes and headers. For most packet-oriented PPP links, this is not an issue since they don't use HDLC or byte stuffing.

Returns

Nothing

ppp_lowerdown

Name

`ppp_lowerdown()`

Syntax

```
void ppp_lowerdown(M_PPP mppp, int pcode)
```

Description

This should be called from the line drivers whenever a connected device terminates the connection. This includes terminations which are the result of an ln_disconnect() request. The PPP layers may attempt to send bytes via ln_putc(), however the line code is free to discard them.

As with ppp_lowerup(), the first parameter is the mppp structure associated with the device when it was initialized. When called from the line drivers, `pcode` should always be the LCP_STATE code.

Failure to call this routine after unexpected disconnection will usually result in PPP being unable to use the line device.

Returns

Nothing

ppp_lowerup

Name

`ppp_lowerup()`

Syntax

```
void ppp_lowerup(M_PPP mppp, int pcode);
```

Description

This routine is called by the FSM after a lower level has completed its initialization. The pcode parameter specifies the next protocol that should begin its initialization.

For the porting engineer, the primary relevance of this routine is that it should be called from the line driver code whenever it detects a change in line state from not connected to connected. A common example of this is a modem in auto-answer mode accepting an incoming call. This callback to PPP will initiate the correct PPP events. In the example of a modem answering, the PPP layer will send packets to begin an LCP link as a server.

Line drivers should pass the mppp structure they where associated with when they were initialized, and should always pass the LCP_STATE code as the second parameter. Other protocol states, such as IPCP_STATE, are considered internal to PPP. They should only be used if you are adding new protocol layer to PPP (for example a new type of authentication) and not for normal porting or device driver authoring.

This call may take a while to complete, and thus should not be called from an ISR or similar protected or time-critical mode. The sending of an initial LCP configuration request (CONFREQ, the beginning of option negotiation) will occur in the context of this call, so the line device should be prepared for a series of calls to ln_putc() before this is called.

Returns

Nothing.

ppp_timeisup

Name

`ppp_timeisup()`

Syntax

```
void ppp_timeisup(void);
```

Description

This routine is provided in `ifppp.c` and is called once per timer tick. It loops though the list of active ppp sessions to determine if any of the ppp timers have expired. If a timer has expired, it calls the timeout function for that ppp session (`mppp->tmo_func()`).

`ppp_timeisup()` must lock the `PPP_RESID` before looping through the list of sessions. On a multitasking system, `ppp_timeisup()` is called in the context of the timer task, and therefore should not block waiting for the lock. It calls `WAIT_NET_RESOURCE` with a timeout of 0. If the lock is not immediately available, it will return and try again on the next timer tick.

Returns

Nothing

prep_ppp

Name

`prep_ppp()`

Syntax

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

Description

This routine is called by the system during device initialization if one or more ppp devices are listed in the `in_devices` table in `userdata.c`. The porting engineer should assure that `in_devices[ ]` contains one entry for each ppp interface that should be created and initialized during system initialization. `ppp_prep()` will initialize each ppp device (interface) listed in the table.

When PPP is used with the InterNiche IP stack, the `ppp_prep()` routines should require little or no change. On non-InterNiche system, it may need extensive rewriting.

This sets the number of static interfaces (`nets[ ]`) to be used for PPP and maps one `M_PPP` structure to each interface.

Note that this does not create dynamic PPP interfaces. These are created by calls to `ppp_create();`

Returns

Returns the number of interfaces initialized.