TECHNICAL NOTE: PORTING INTERNICHE SLIP

1. THIS TECH NOTE

This document discusses the steps a porting engineer may find necessary as InterNiche's SLIP implementation is ported to a new target platform, it discusses the tunable parameters and the structural and file system organization of the product.

2. ARCHITECTURAL OVERVIEW

Most of the SLIP code is organized into files that bear the name of the layer or module implemented. Most of the SLIP definitions are in the header file **slip.h**. Each of the files is described in the pages that follow.

2.1 The slip_errors Structure

Described in RFC 1055, SLIP is one of the simplest and smallest RFCs and was a precursor to the more versatile PPP protocol. InterNiche's SLIP implementation maintains the simplicity of the RFC by being a stateless implementation and as such does not require any structures. It supports only one SLIP interface for the host and hence uses global data buffers for the receive, transmit, SLIP escape byte stuffing and SLIP escape byte stripping operations. The only structure used is the slip_errors structure as given below, which logs the different kinds of errors detected on the line.

```
struct slip_errors
{
  unsigned   serr_noends;      /* didn't find SL_END where expected */
  unsigned   serr_overflow;    /* frame overflowed buffer */
  unsigned   serr_connect;     /* lower level connect failed */
  unsigned   serr_nobuff;      /* buffer alloc failed */
  unsigned   serr_noframe;     /* got char outside of SLIP frame */
  unsigned   serr_iphdr;       /* SLIP data was not IP packet */
  unsigned   serr_c0noise;     /* Line noise or Peer Bug introduces spurious 0xc0 */
};
```

2.2 Line Drivers

InterNiche SLIP accesses the underlying network hardware by means of the line driver API. Each type of device (UART, Modem) which is to carry SLIP traffic must implement these calls, which are described below. 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 SLIP code whenever a link comes online, such as a modem receiving an incoming call. In order to support Hayes compatible Modem dialing, a modem line driver is provided which assumes a simple byte oriented UART driver beneath it.

2.2.1 com_line Structures

The structure **com_line**, defined in the header file **h/comline.h**, manages the line drivers. The SLIP implementation uses a single **com_line** structure as the implementation supports only one

SLIP interface. The **com_line** structure describes the interface between an upper layer (in this case SLIP) and a lower layer, such as modem code. 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**. 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 */
```

Note that another **com_line** structure may exist below the lower layer, for example to map a modem dialer to a UART driver.

3. PORTING STEP BY STEP

This section outlines the steps needed to port the InterNiche SLIP code into the software of a pre-existing InterNiche IP stack. If you are implementing SLIP at the same time as the IP stack, simply defining **USE_SLIP and USE_COMPORT** in **ipport.h_h** and including the SLIP sources in your **makefile** (or build script) will do most of the work for you. In any case, porting engineers should not need to modify any files for SLIP at all, as all the SLIP files can be 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, they should FIRST discuss the intended modifications with the InterNiche technical support staff.

We can either suggest an alternative or modify our sources 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_slip()** and **uart_check()** 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 rely on the InterNiche stack to initialize the driver and SLIP code. Other IP stacks will need a glue layer, responsible for mapping their initia lization, send and close calls to SLIP. SLIP's calls are those described for an InterNiche Net Structure as described at length in the InterNiche TCP/IP NicheStack Reference Guide or Porting Guide.

SLIP can be customized for your particular application in a couple of areas. This includes configuration settings such as:

1. The maximum size of the SLIP packet to be sent on the line. The current default value is 1006, RFC 1055 recommends that a host is capable of accepting a SLIP frame of size 1006 and also should not transmit a frame greater than 1006 bytes. In both instances the 1006 bytes

does not include the SLIP escape byte characters or the SLIP headers. If required the value can be changed by modifying the macro **SLIPSIZ**, defined in **slip.h**

2. If **SLIPSIZ** is modified then **SLIPBUFSIZ** should also be examined. **SLIPBUFSIZ** is the maximum size of the SLIP internal Buffer. This has to accommodate 1006 bytes excluding the ESCAPE characters. So assuming the 1006 frame required by the RFC is full of escape characters then the min Buffer Size to guarantee reception and transmission would be 2012 bytes. For the sake of rounding, 2048 has been selected as the default value. **SLIPBUFSIZ** is defined in **slip.h**

3. **SLIPHDR_BIAS** needs to be set to 3 to force alignment of pkt->nb_prot on a 4 byte boundary so that IP and TCP headers meet alignment requirements of architectures such as ARM, NEC, etc. The current value is set to 3 to force alignment by default on all builds regardless of the architecture. Hence, this value also can be left as is. **SLIPHDR_BIAS** is defined in **slip.h**

4. The **FAST_SLIP** feature is enabled by default in **slipport.h**. This makes use of the system call memchr() to provide a slightly more optimized implementation. If your toolset or compiler does not provide the memchr() library routine, disable this option.

5. The **SLIP_PCUART** macro (defined in **slipport.h**) selects the lower layer type. This macro enables SLIP to hook to the ln_*() line drivers described in the following sections and work over on a direct connect NULL modem cable. Disabling this and defining **SLIP_MODEM** is the first step in the process to configure SLIP with a Modem Line Driver. Please note that InterNiche SLIP is provided with SLIP_PCUART. If you want to run over a Modem Line Driver, please request sample Modem Dialer code from InterNiche, and then port the SLIP implementation to use the dialer code.

3.1 Source Files

As provided, the SLIP source code is just two **.c** source files and two **.h** (include) files. These are called the SLIP "portable" or "port-independent" source files, and should not need to be modified for a simple SLIP port.

The portable **.c** source files are:

slipif.c - core SLIP implementation and interface into Interniche IP Stack.
slip.c - SLIP escape byte stripping and stuffing routines.

The portable header files are:

slip.h - core SLIP defines and configuration file.
slipport.h - SLIP lower layer select configuration file.

The best first step is usually to compile the code as received from InterNiche. A **readme.txt** file for 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 SLIP, and you will have the opportunity to step through the SLIP 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.

Please refer to **build_tn.pdf** for information about building the sources.

The SLIP code includes (usually by including **ipport.h_h** which include **nptypes.h**) defines for the data types shown in the following example:

```
typedef     unsigned char u_char;       /* 8 bit unsigned */
typedef     unsigned short u_short;     /* 16 bit unsigned */
typedef     unsigned short unshort;     /* duplicate */
typedef     unsigned long u_long;       /* 32 bit unsigned */
typedef     int       bool;             /* another popular type extension */

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

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

For most compilers you can use these defines exactly as they appear in the reference package. If your compiler has unusual type requirements your may need to modify these.

3.2 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.

3.2.1 Modem Unit Numbers

In order to support multiple modems (such as if a PPP Stack is also functional on the host), 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 which only support 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 can resume to try sending 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 SLIP 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. This thread also supports many other timer functions throughout the InterNiche system, and 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 received bytes the UART has buffered characters while the thread was sleeping. Since the thread may sleep for up to half a second, then 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. In a situation like the one just described, it is quite likely that no SLIP packets will ever be received intact. The UART might just as well not exist.

This situation can be detected by the presence of large numbers of “serr_noframe”(received characters outside of SLIP frame) or “serr_iphdr” (SLIP data was not IP packet) errors in the slip_errors statistics. 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
- Can 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.

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 **nedxt** 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 SLIP 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.

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.

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.

NAME

uart_stats()

SYNTAX

int uart_stats(void * pio, int unit);

DESCRIPTION

Display UART statistics to the output device **pio** 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.

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 SLIP 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**.

3.3 Line Drivers

Each type of line device supported by the SLIP code will require a "line" driver implementation. Line drivers are provided for Hayes modems and several UART's. If you want to use any other device, you will need to create your own driver and attach it to the SLIP 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 SLIP unit */
  int   lower_unit;           /* legacy ID for lower (UART level) drivers */
};
```

The SLIP implementation uses a single **com_line** structure internally as the current implementation supports only one SLIP implementation per host. The pointers to the driver routine are set up when the SLIP interface is prepared to be initialized later on (see **prep_slip()**), and a pointer to the **UNIT** 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). When the SLIP 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 **prep_slip().** 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. SLIP will not re-enter the routines or assume any sort of time-out.

Devices have to be character (byte) oriented. Character oriented devices, such as UARTs and modems, should deliver received characters to SLIP by passing them to the line device's **ln_getc()** function. The function pointer is set by the SLIP code in **prep_slip()**, and usually

points to **slip_rxbyte()**. 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 SLIP, such as PPP.

The routines are defined in the following pages.

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 SLIP 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

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; SLIP does not take any action based on it.

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, SLIP 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.

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_prot - points to data to send (SLIP header)

pkt->nb_plen - length of data at **nb_prot**

The current SLIP implementation does NOT use this feature of the Line Drivers, which is documented here only for informational purposes.

RETURNS

Return values are the same as **ln_putc()**.

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 **prep_slip()**. Data for IP should be delivered to SLIP via this routine on UART-like devices.

RETURNS

Returns 0 if OK, else negative (**ENP**_) error code.