InterNiche Porting Kit Reference Manual

Interniche Legacy Document

Version 1.00

Date: 10-May-2017 15:46

Table of Contents

1 Introduction

The harness is the name given to a stack-less project used to assist in developing a new target port for NicheStack. By using this tool, it is possible to develop a complete target port without involving the complexity of the TCP/IP stack, and when your port is ready it will simply be a matter of linking it into your complete network-enabled application.

A Port is composed of three major portions:

1. The controller or processor executing the code;
2. The Operating System in control of scheduling threads;
3. The toolchain and device library providing the runtime environment.

Any or all of these components may need to be developed by the harness user or can safely be imported from other InterNiche-based projects. Before delving too far into use of the harness, be sure to ask InterNiche if any of the desired portions are already available. Most of this document is concerned with the chip portion, but the initial steps will discuss the needs of the OS and of the toolchain.

2 PORT

A Port is composed of three major portions, the chip running the code, the OS in control of scheduling threads, and to toolchain and device library providing the runtime environment.

PACKET

A PACKET structure. This is a buffer containing a start and a length pointer for both the buffer as a whole, and the portion contining data. Also included is a pk_next/prev pointer to create a chain of buffers making up a frame, and the total length of that frame (first PACKET only).

FRAME

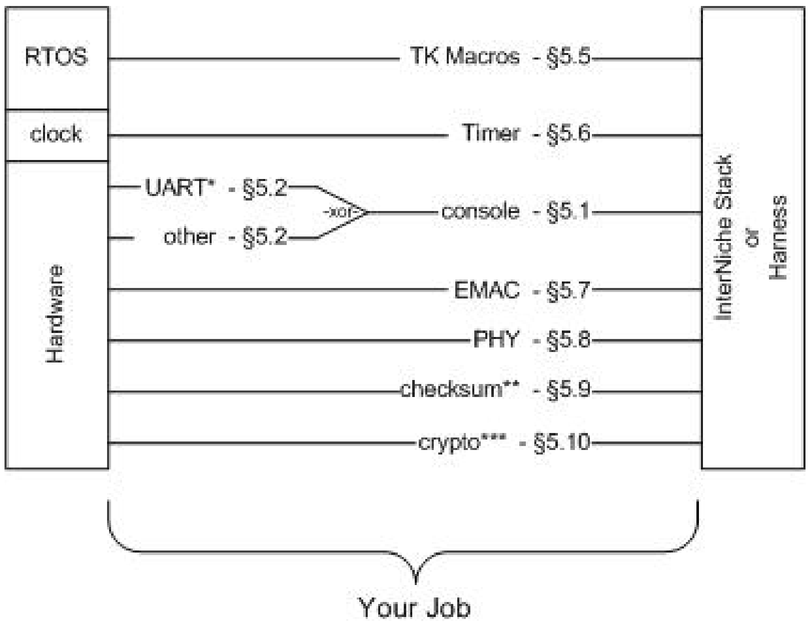
A chunk of data that goes out the network interface as a unit. e.g. 802.3 headers + payload + FCS

FCS

Frame check sequence. Not calculated by the stack, driver must do this if not done by hardware.

3 Overview

As mentioned earlier, a port has three logical components, the operating system interface, the hardware interface and the toolchain used to create the actual executable image. This document will guide you through creating a port without involving the actual NicheStack, and will explain how to use the supplied framework to verify that your port is operational.



The Harness provides two functions. First, it is the instrumentation of the port's entire API. It provides buffers in exactly the same was as does the NicheStack. Timers execute exactly the same way. Likewise serial and console i/o, OS task switching cryptographic support, and interrupt handling.

The Harness is used to assist in bringing up the port specific portions of an InterNiche stack on new hardware/os/toolchains while keeping the complexity and indeterminism out of the process as much as possible. To this end most of the stack is stubbed out, and a simple menu interface is provided to interact with the stack.

Along with the harness code, a port directory is provided with skeletons of the required files. Some of these are expected to be used as is, others will require the engineer to complete functions before certain features work, these are identified within the harness source code as `TODO:`.

4 Harness Menu Interface

The harness menu interface consists of a list of numbered commands, some of which will ask for additional parameters when executed. These prompts for additional information will provide a default value in square brackets, just pressing enter will use the default.

A full menu depends on features being enabled in libport.h, the section discussing implementations will note what they are. Pressing `0` at the prompt will display a menu of commands similar to the following.

```
    0.    display menu
    1.    sleep cticks
    2.    sleep pticks
    3.    reset
    4.    cksum test
    5.    critical section test
    10.   cfg eth
    11.   cfg buf
    12.   init eth
    13.   reset eth
    14.   close eth *
    15.   trace pkt recv
    16.   pkt recv
    17.   send pkt
    30.   MD5 validation
    31.   SHA1 validation
    35.   AES validation
    39.   RAND validation
    50.   dump queues
    51.   dump stats
    52.   dump phy registers
```

The harness has a few items to help with testing portions of the driver. First is the open call, this calls the setup routines. Once you can pass initialization, you can proceed in one of two directions,

Rx monitoring will show a slightly decoded form for each packet received, the menu interface will return instantly, but the packet decodes will continue until you invoke the same item again.

Tx will send hand crafted ping packets to a pre-determined MAC/ip address. These should be well formed enough for the ping'd host to respond and you should see them in the rx path (on the terminal if you have rx monitoring, in driver printf's if you have not gotten that far yet). It needs mention that the harness will NOT respond to ARP requests, so the destination needs to have a static ARP entry for the target. The packets that are crafted for this option are highly fragmented, so will provide a good test of chaining and odd size /alignments of the buffer start/lengths.

The eth setup option will allow you to change the mac/ip of you and your destination at runtime. or edit the values in `harness_mod.c`

With ethernet done, there is an option to provide an optomized version of the Cksum generation routine. This routine is called asm_cksum (weither it is or not in asm) and takes a buffer and a count of `uint16_ts` to checksum. There is a harness option to verify correctness and also provide performance measures of the checksum operation.

The harness critical section test simply reads the clock, enters a critical section, spins for a few seconds, exits the section, then compares the clock. If no more than one tick has happened, you pass.

5 Step-by-Step

5.1 Begin With Something that Works

The first step is to create a very basic application that can be downloaded to your hardware without involving any InterNiche code that way you will begin developing your port from a known, working point. We recommend using some form of "blinking light" program as it will demonstrate that the main chip-specific initialization, clock and interrupt setup are all working. This program should not include any RTOS involvement, as adding that complexity at this point may cause difficulty a few steps from now. From this point, we will add and develop the remaining pieces. This starting point is usually provided by the toolchain vendor.

5.2 Console

The next step is to address the console. The console interface is split into a upper layer API and a lower level driver that provides UART level access. If the target has a console other than a UART then you should replace the default console interface provided by console.c, otherwise leave it as provided and implement the UART driver and let the console take care of what details it can.

The Console API consists of three routines: `int kbhit()`, `char getch()`, and `int putchar()`.

If you are porting a new toolchain then now is the time to correct the definitions in toolport.h and also how do wire in the libraries printf/output redirection The output redirection should be implemented in terms of `dputchar` or `uart_putc`.

Next, you should bring the following files into your project:

- `console.c`
- `uart_util.c`
- and the empty `uart_drv.c`

and add a loop to main:

```
system_init();
while(1)
{
   if (kbhit())
   {
      dputchar(getch());
   }
}
```

5.3 UART basic

The implementation provided with the Harness uses calls to the UART driver's data API using the UART id of `CONSOLE_UART`. There is an alternate implementation that uses Unix-like tty access.

If the UART version of the console is to be used, then an implementation of the UART driver is required.

The UART interface is best thought of as having an "upper" and "lower" interface, with the upper interface controlling the UART data, ane the lower interface controlling the actual device

The UART driver is a buffering IO stream that is sutable for console use, and for PPP. Each device is provided with two buffers, one for input and one for output. These buffers are accessed by the data API to provide higher level access to the ports, and by the device API on the hardware side to pass to the external port. Console access ports can get by with just using the update/kick API to receive/send, but PPP capable links should use some form of DMA and interript logic to keep up with the character stream.

The UART data API:

The UART data API consists of three routines and should not need modification:

- int uart_gotc() - returns a 1 or 0 depending upon whether there is, or is not a character to read.
- uart_getc() - get one character from the UART (buffered).
- uart_putc() - write one character to the UART (buffered).

The UART device API:

The UART data API consists of four routines that must be implemented by the porting engineer:

- USART_Configuration() - pin and power/clock setup for UART.
- uart_init() - initialize an instance of the UART driver.
- uart_update() - check for UART input, update input buffer.
- uart_kick() - transmit character(s) in buffer now (may return as long as data will make it out without more interaction).

Fill in the UART portions using `hwport.h` if you intend to support multiple board layouts. Also fill in `uart_drv.c` for `uart_init`, `uart_update` and `uart_kick` and `sysinit.c`'s `USART_Configuration()` to setup pins.

5.4 malloc and free

If the standard library provided by your toolchain vendor does not include `malloc()`/`free()`, you can use the provided `alt_memory.c` file to provide them. These will be invoked from `in_memory.c`, and to avoid future confusion you might consider renaming them. The `alt_alloc()`/`alt_free()` implementations work for many situations and also serve as a clean and simple starting point Even if your OS provides them, we recommend using these simple ones for now, and when you get to the OS stage they can be replaced.

Now it is possible to remove the `main.c` from your sample project and add the files in from the InterNiche distribution. Building your project should now produce an image that displays the menu and allows you to type '`0`' for help

5.5 UART - optional

PPP

If PPP is desired, `uart_drv.c` should be revised to handle higher through channels. Either use flow control pins (recommended) or implement a software version of it in the `uart_*` routines. The use of DMA to provide timely hardware interaction is also recommended as is using a buffer that can hold at least a full-sized packet.

5.6 OS

The OS API

The 'SuperLoop' OS is provided as a reference implementation, if that is the intended target, then you need not perform this step. To port to an OS you need to fill in osport.h and osporttk.c. Tasking routines first, TK_CREATE, TK_SUSPEND, TK_RESUME, TK_DELETE, TK_YIELD, TK_SLEEP and then TK_INIT_OS and TK_START_OS. The InterNiche porting API was designed to easily map to most operating systems but should you have difficulty, contact Support@HCC-Embedded.com. After getting the tasks sorted out, work on semaphores and mutexes. Semaphores need only be binary, but may count if desired, where mutexes should be non-nesting.

Critical sections entry points (located in `inport.c`) now need to be implemented in the routines `crit_section_enter()` and `crit_section_exit()`. Critical sections should nest, so the section should not restore interrupts until all sections have exited. This is done with a counter in the provided examples, and unless the OS provides this, they will be acceptable.

The OS API consists of the following entry points which should be mapped to the facilities underlying your chosen RTOS.

ENTER_CRIT_SECTION

critical sections - disable irq

EXIT_CRIT_SECTION

critical sections - re-enable irq (when unnested)

LOCK_NET_RESOURCE

Acquire a mutexTK_BLOCKWait for something to happen

TK_CREATE

Create a new task.

TK_DELETE

Delete a task

TK_SIGNAL

post to a semaphore

TK_SIGWAIT

wait for a semaphore to be posted to

TK_SLEEP

Wait for n seconds (calls TK_BLOCK)

TK_SUSPEND

Stop a task

TK_YIELD

Relinquish CPU, remain runnable

UNLOCK_NET_RESOURCE

Release a mutex

WAIT_NET_RESOURCE

Try to acquire a mutex, return if not available

The OS Port: osport.h, osporttk.c

The provided skeleton is actually a fully operational OS implementation. Every macro/function needs to be updated if a different OS is required. The harness utilizes three tasks, console, timers, and packet demux. The full stack has equivilent tasks, plus some protocols have a task of there own to maintain state.

The provided OS is a copy of SuperLoop, in effect just calls the 'tasks' in a forever loop. It will work while you develop the chip drivers more completely, or you can replace it now with a port to the target OS.

5.7 Time

Time is kept in a `uint32_t`. Both `cticks` and `pticks` are held in 32 bit words and rollover is to be expected; at 20Hz (ctick default) this roll over is 6.8 years, 1kHz for pticks is ~50 days. Even so, there are macros for time comparisons.

If your OS provides a system clock, you need simply point `get_cticks()` and `get_pticks` at it. `CTICKS` should have a value incrementing at TPS Hz (nominal 20Hz) while `pticks` may be whatever is native. Use `intimer.c` `clock_init()` to initialize what is needed. Any ISR code should also go in here.

clock_init

Initialize time base.

get_cticks

TPS ticks since system init (20Hz)

get_pticks

PPS ticks since system init (system dependent speed >= 100Hz)

The harness execution should now be able to execute the sleep tests with the pause before returning to the menu interface being approperate to the specified delay.

5.8 Ethernet MAC

Network frame data is held in buffers pointed to by PACKET strutures. PACKETs are reserved from the heap at startup time and are allocated from a pool by PK_ALLOC. In order to reduce resource requirements the size of the PACKET buffers is small, normally 128 or 512 bytes, while ethernet frames may be larger. This is handled by chaining many PACKETS together to produce a single frame of data. There are a number of functions for manipulating data within the frame that will deal with PACKET crossing and updating the recordkeeping within them. By convention, it is the responsibility of each protocol layer to maintain the integrity of the following fields within each PACKET:

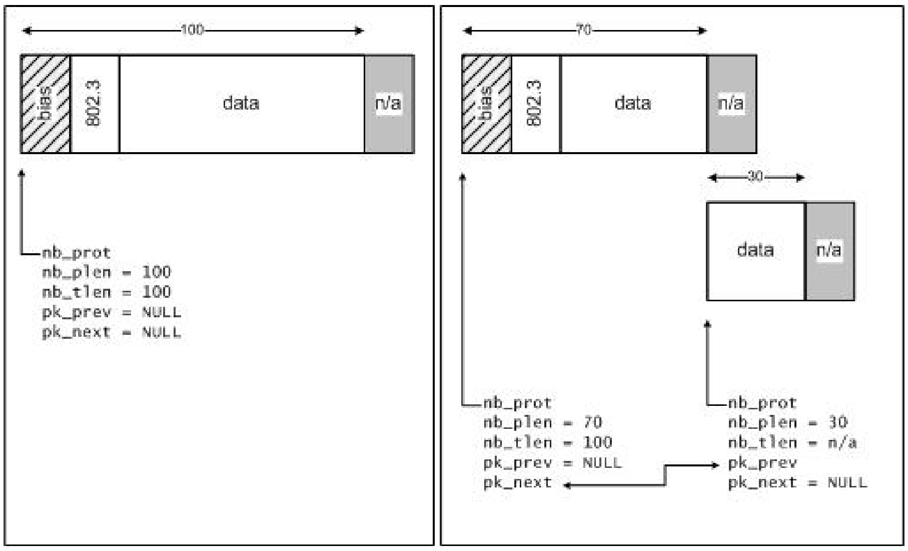
nb_tlen	The total length (in bytes) of data held in the chain of PACKETs. This required value is only meaningful in the **first packet of the chain**.
nb_plen	The number of data bytes in the current buffer.
nb_prot	The address of the first byte of data in the current buffer.
pk_next, pk_prev	These values constitute the doubly-linked list of PACKET in the chain.
nb_blen	(read only) The size of the buffer as allocated. This value must be **at least 64 bytes**.
nb_buf	(read only) Address of the allocated buffer.

This driver is enabled by setting USE_EMAC in libport.h. Also relivant are USE_EMAC_UTIL and USE_GENPHY.

The file emac_drv.c is where the example provides entrypoints. Mostly it consists of a few sections of init, a close, a hand to hardware for TX. a from hardware to PACKET rx, an interupt function, and phy read/write. The actual driver interface is mostly defined in emac_util.c which contains all the boilerplate code that does not change.

The driver is given some pre-allocated PACKETs into which it will receive data. When these get used by the rx logic, they should be replaced by using PACKETS in the togetq that is maintained by the emac_util.c portion of the driver. This allows the rx PACKETS to be handed to the recieve queue, and the driver to have something to replenish the space to hand back to the hardware without invoking the PACKET allocation routine from ISR context, allowing that allocator to be simplified. Similar, for TX completion ISR, put the finished PACKETS into the 'toputq' and they will get free'd in the mainline context soon.

With an ETHHDR_BIAS of 2, a properly formed packet chain for an inbound 98-byte packet could look like either of the following:



Functionally Equivalent PACKETs

In order to keep the ip addresses in the IP header aligned to a word (4 bytes) boundary, it is useful to pad the ethernet headers from 14 bytes to 16 by shifting the start of the buffers forward two bytes. The ETHHDR_BIAS define in libport.h is used for this and is part of the value of ETHHDR_SIZE. Some hardware needs to be told the two byte shifted value, others are configured to do the shifting on their own and require the normal start of buffer address. This bias is expected to be included in the pointed to nb_prot for frames being sent, and pointed past along with the ethernet headers on the received frames being passed to the stack.

The routine `eth_prep` points these to functions that call emac_drv entrypoints

- XXX_prep - Device initialization routine
- net->n_init - Initialize the hardware
- net->n_close - De-initialize the hardware
- net->n_stats - network statictics.
- net->pkt_send - send a 802.3 packet
- net->n_refill - Replenish togetq and free toputq

The file `emac_drv.c` (called by emac_utils.c) should contain the following functions:

- emac_rxtx_init - Prepare descriptors for hw
- emac_mac_init - Do the work of initializing the hardware
- emac_hw_init - initialize pins and isr
- emac_close - deinit isr and hw block
- emac_send - put frame worth of PACKETs in descriptors
- emac_phy_read - read PHY register
- emac_phy_write - write PHY register
- eth_setlink - set link speed
- emac_phy_status - set phy link status
- emac_core_enable - associate hardware with an ethernet instance

5.9 PHY

The file `phy_generic.c` contains the following functions that should not require changing:

- emac_phy_init - PHY setup.

The file `phy_XXX.c` should contain the following functions:

- phy_link_detect_init - PHY specific initialization and setup for link change irq

A generic PHY implementation is provided in phy_generic.h with one exception, an interface is exposed to initialize the PHY to cause an interrupt when link is lost or gained, as well as any customer specific PHY init features that are desired. A usable stub for this is in phy_noirq.c and phy side setup for well known PHYs is also provided, but they will all need customazation to hook up the irq to the correct source.

5.10 Checksum - optional

`asm_cksum()` is an accelerated checksum routine that will be used when `C_CHECKSUM` is NOT defined in `ipport.h`. This routine follows the interface of `cksum()` bud adds the qualifier that the data is 16-bit aligned.

5.11 Crypt

Hardware assisted crypto is enabled with `USE_HW_CRYPT`. There is an example `sec_drv.c` that contians entrypoints for the known crypt block cyphers and hashes. Fill in the functions to perform the operation using HW. All of these interfaces are expected to be blocking, so doing a spin wait or yield loop polling hardware is acceptable.

The harness has options to run a single block thru the cyphers and hashes and verify correctness of the answer. The vectors were taken from NIST documents.

6 API

6.1 dputchar

API Name

`dputchar() - Send a character to the console`

Syntax

```
void dputchar(int chr);
```

Description

The InterNiche CLI routines call `dputchar()` in order to display a character on the target system display or monitor port. If such output is not desired, `dputchar()` can be implemented as a no-op. Its parameter is an ASCII character that should be displayed on the target system display or monitor device.

`dputchar()` should perform newline expansion. If the value of `chr` is an ASCII newline character (`0xa`) then a newline followed by a carriage return should be output to the display device.

Returns

Nothing.

6.2 getch

API Name

`getch() - Get character from console`

Syntax

```
int getch(void);
```

Description

`kbhit()` and `getch()` are used together to effect CLI input. The stack code calls `kbhit()` to determine if a character is available and then if a character is available, calls `getch()` to return the value of the character. **`getch()` should never block for user input**.

Returns

If a character is available at the CLI or system monitor device, `getch()` returns the ASCII value of that character. Its return value is undefined if no character is available.

6.3 kbhit

API Name

`kbhit() - Pool for character ready from console`

Syntax

```
int kbhit(void);
```

Description

`kbhit()` should return a non-zero value if a keystroke has been entered by a user at the CLI of the target system. It should not dequeue the character itself from the input device, rather the return value from `kbhit()` should simply poll the device to determine if a character is present. The entered character is retrieved using the `getch()` function.

Returns

`0` if no character had been entered at the input monitor device, non-zero if at least one character is available.

6.4 console_only

API Name

`console_only() - Stop all threads except console`

Syntax

```
void console_only(void *pio, bool_t dumpsystem)
```

Parameters

pio	Handle for output. If NULL, output goes to stdout.
dumpsystem	Non-zero means call the `dumpsysinfo` API before suspending tasks.

Description

Suspends all tasks, except the console. If dumpsystem is non-zero, it will call the `dumpsysinfo` API before suspending tasks. During debugging, an engineer could call this API when a special condition occurs, e.g., a dtrap. This API is only available when `NPDEBUG` is defined.

NOTE: The system cannot be returned to a normal state following this API.

Returns

- None

6.5 ENTER_CRIT_SECTION, EXIT_CRIT_SECTION

API Name

`ENTER_CRIT_SECTION() - Enter critical section`

`EXIT_CRIT_SECTION() - Leave critical section`

Syntax

```
#define ENTER_CRIT_SECTION
```

```
#define EXIT_CRIT_SECTION
```

Parameters

None

Description

These two entry points should be designed to be paired around sections of code that must not be interrupted or pre-empted. Usually, `ENTER_CRIT_SECTION()` should save the processor interrupt state and disable interrupts, whereas `EXIT_CRIT_SECTION()` should restore the processor interrupt state to what it was before the most recent call to `ENTER_CRIT_SECTION()`.

See the detailed discussion of these macros see Critical Section Method.

Returns

These return no meaningful value.

Note

Nested calls to ENTER_CRIT_SECTION() and EXIT_CRIT_SECTION() **must** be supported.

6.6 cksum

API Name

`cksum() - Calculate buffer checksum`

Syntax

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

Parameters

```
char *buffer /* pointer to buffer to checksum */
```

```
unsigned word_count /* number of 16 bit words in buffer */
```

Description

Returns 16 bit Internet checksum of buffer. Algorithms for this are described in RFC1071.

NOTE: A portable C language version of this routine is provided with the demo packages, however TCP implementations can spend a significant portion of their CPU cycles in the checksum routine. This routine is described here to encourage porting engineers to optimize their ports by implementing their checksum routines in assembly language. An Intel x86 assembly language checksum routine is also included which can be used on Intel processors as is. Versions for other CPUs are widely available - contact us if you need help finding one.

Returns

The 16 bit checksum.

6.7 clock_init

API Name

`clock_init - Start the NicheStack clock`

Syntax

```
void clock_init(void);
```

Parameters

None.

Description

This sets up a periodic call to the function in_tick_hook() at a rate of 100Hz (defined as PPS). It is possible that the OS timer is faster than this, and there is a provision for skipping n calls before ticking.

If the RTOS does not provide a time base then the port must provide one and handle the irq routing to get in_tick_hook called.

If the NicheStack clock is disabled, reset all NicheStack clock variables to their initial values and enable the NicheStack clock. If the NicheStack is already enabled, calls to `clock_init( )` do nothing. A call to `clock_c( )` to disable the NicheStack clock is required before the clock can be reenabled.

Returns

Nothing

6.8 emac_close

API Name

`emac_close() - Shutdown Ethernet device`

Syntax

```
int emac_close(int index)
```

Parameters

`index`	interface number in the nets array

Description

Disable the ethernet hardware, and free any allocated resources

Returns

Success(0) or failure (1)

6.9 emac_core_enable

API Name

`emac_core_enable - Enable HW block for ethernet`

Syntax

```
int emac_core_enable(IN_ETH eth)
```

Parameters

`eth`	Ethernet control block for the intended inteface

Description

Enable the hardware block for the ethernet module, setup any pointers to the block that may need to be remembered in the IN_ETH structure.

Returns

Success(0) or failure (1)

6.10 emac_hw_init

API Name

`emac_hw_init - Start ethernet interface traffic`

Syntax

```
int emac_hw_init(IN_ETH eth)
```

Parameters

`eth`	Ethernet control block for the intended inteface

Description

Enable the hardware to start transfering packets, inluding enabling the interrupts. This function is called from the common ethernet initialization code.

Returns

Success(0) or failure (1)

6.11 emac_mac_init

API Name

`emac_mac_init() - Ethernet interface config`

Syntax

```
int emac_mac_init(IN_ETH eth)
```

Parameters

`eth`	Ethernet control block for the intended inteface

Description

Initialize the hardware block for operation. This function is called from the common ethernet initialization code.

Returns

Success(0) or failure (1)

6.12 emac_phy_read

API Name

`emac_phy_read() - Read PHY register`

Syntax

```
unsigned short emac_phy_read(IN_ETH eth, unsigned phyaddr, unsigned phyreg)
```

Parameters

`eth`	Ethernet control block for the intended inteface
`phyaddr`	address of the PHY
`phyreg`	address of the register to read

Description

Read a register from a ethernet PHY attached to this ethernet block

Returns

16 bit word read from PHY

6.13 emac_phy_write

API Name

```
emac_phy_write() - Write PHY register
```

Syntax

```
void emac_phy_write(IN_ETH eth, unsigned phyaddr, unsigned phyreg, const
unsigned short data);
```

Parameters

`eth`	Ethernet control block for the intended inteface what it does
`phyaddrvar1`	address of the phy what it does
`phyregvar2`	address of the register to write what it does
`data`	Value to write to the PHY

Description

Write a register from a ethernet PHY attached to this ethernet block.

Notes/Status

Returns

Nothing

6.14 emac_rxtx_init

API Name

`emac_rxtx_init()- Setup ethernet buffer usage`

Syntax

```
int emac_rxtx_init(IN_ETH eth)
```

Parameters

`eth`	Ethernet control block for the intended inteface

Description

Initialize the rx and tx descriptors and packet queues. This function is called from the common ethernet initialization code.

Returns

Success(0) or failure (1)

6.15 emac_send

API Name

```
emac_send() - Process the outgoing packet queue
```

Syntax

```
void emac_send(IN_ETH eth);
```

Parameters

`eth`	Ethernet control block for the intended inteface

Description

Invoke the send logic to see if any of the queued packets are able to be passed to the hardware. This routine is called from the common ethernet code from within the function called by `net->pkt_send`.

Notes/Status

Returns

Nothing

6.16 eth_prep

API Name

`eth_prep() - Setup ethernet nets structure`

Syntax

```
int eth_prep(int index)
```

Parameters

`index`	interface number in the `nets` array

Description

Prepare the ethernet interface structure for this device, including populating the interface function pointers and flags for operation.

Returns

success(1) or failure(0)

6.17 eth_setlink

API Name

`eth_setlink()- Set ethernet link status`

Syntax

```
void eth_setlink(IN_ETH eth, unsigned phyaddr, int speed, bool_t duplex)
```

Parameters

`eth`	Ethernet control block for the intended inteface
`phyaddr`	address of the phy
`speed`	speed of connection
`duplex`	Full duplex(true) or half (false)

Description

Inform the network driver about a change in link status. This is called from the phy link change ISR and from the init routines.

Returns

Nothing

6.18 get_cticks

API Name

`get_cticks() - Get slow timer tick count`

Syntax

```
uint32_t get_cticks(void)
```

Parameters

None.

Description

Counts time since power up. Frequency is TPS (default 20Hz)

Returns

Returns 32-bit tick count of time since power up.

6.19 get_pticks

API Name

```
get_pticks() - Get fast timer tick count
```

Syntax

```
uint32_t get_pticks(void)
```

Parameters

None.

Description

Counts time since power up. Frequency is PPS (default 100Hz)

Returns

Returns 32 bit tick count of time since power up.

6.20 LOCK_NET_RESOURCE, UNLOCK_NET_RESOURCE

API Name

`LOCK_NET_RESOURCE() - Resource access lock`

`UNLOCK_NET_RESOURCE() - Resource access unlock`

Syntax

```
void LOCK_NET_RESOURCE(int resID);

void UNLOCK_NET_RESOURCE(int resID);

void WAIT_NET_RESOURCE(int resID, int timeout);
```

Parameters

Any of the `xxx_RESID` constants.

Description

See description of Net Resource Method.

Returns

Nothing.

6.21 TK_BLOCK

API Name

`TK_BLOCK - Relinquish the CPU to another task`

Syntax

```
void TK_BLOCK();
```

Parameters

None.

Description

The macro is called by the current task when it has no more immediate work to do. In most systems, this macro is equivalent to TK_YIELD(). Execution of the current task is stopped and execution of the next task that is ready to run is started.

Notes/Status

See TK_YIELD() for further discussion.

Returns

Nothing

6.22 TK_CREATE

API Name

`TK_CREATE - Create a task`

Syntax

```
int TK_CREATE(void (*code)(void *), char *name, int stack, void *param, unsigne
```

Parameters

code	Pointer to the function to be called when the task is started. This function's prototype is: `void code(void *param)` In a tasking environment, this function should never return.
name	`NUL`-terminated string that is the name of the task.
stack	An integer specifying the size of the task's stack in bytes.
param	The parameter passed to the '`code`' function when the task is started.
prio	An unsigned integer specifying the task's execution priority. If the RTOS does not support task priorities, '`prio`' can be zero.
taskp	A pointer to a variable of type `TASK *`. If '`taskp`' is not NULL, the ID of the created task will be stored in '`taskp`'.

Description

`TK_CREATE()` is called to create a task. This may include allocating a task structure, a stack, or other resources for the task. Note that in some tasking systems the task structures and stack memory are statically declared and only need to be activated, while in others, such as NicheTask, the tasks and stacks are allocated from the heap. After the task is created, it is left in the `SUSPENDED` state; `TK_RESUME()` must be called to set the task to run.

Tasks may be started by the system at any time after creation. Tasks should be coded to test for any required resources or conditions as they start executing. An example of this is the netmain_mod.c application tasks, which test the global variable `iniche_net_ready` before commencing network I/O.

Returns

`TK_CREATE` returns `ESUCCESS` if the task was successfully created, and `EFAILURE` otherwise.

6.23 TK_DELETE

API Name

`TK_DELETE - Delete a task`

Syntax

```
void TK_DELETE(TASK tk);
```

Parameters

tk	Task ID of the task to delete. A value of `TK_THIS` is equivalent to the ID of the calling task.

Description

Terminates execution of the specified task and deletes the task's control structure and stack. If the task is self-destructing, execution will continue with the next task that is ready to run.

Returns

Nothing

6.24 TK_SIGNAL

API Name

`TK_SIGNAL - Signal a task from another task`

Syntax

```
int TK_SIGNAL(IN_SEM s);
```

Parameters

s	s is a semaphore object.

Description

When a semaphore is signaled, all tasks that are waiting for the signal (see `TK_SIGWAIT()`) are set ready to be run. `TK_SIGNAL()` is similar to `TK_RESUME()` except that a signal can occur before a task is ready to wait for it. In that case, the signal is recorded, and the task's call to `TK_SIGWAIT()` will return immediately.

Interrupt handlers should use `TK_SIGNAL_ISR()` rather than `TK_SIGNAL()` to wake up a task due to the asynchronous timing between the calls to `TK_SIGNAL()` and `TK_SIGWAIT()`.

Returns

`ESUCCESS` if the signal was recorded successfully and `EFAILURE` if there is an error recording the signal.

6.25 TK_SIGWAIT

API Name

`TK_SIGWAIT - Wait for a signal`

Syntax

```
int TK_SIGWAIT(IN_SEM s, long timeout);
```

Parameters

s	A semaphore object.
timeout	The number of cticks to wait for the signal to occur.

Description

Suspends the calling task until the specified semaphore is signaled or until the specified number of cticks has elapsed. If the semaphore has already been signaled, `TK_SIGWAIT()` returns immediately.

Returns

`ESUCCESS` if the signal was received before the timeout elapsed, `EFAILURE` if there was an error in signaling the semaphore and `TK_TIMEOUT` if the timeout elapsed before the signal was received.

6.26 TK_SLEEP

API Name

`TK_SLEEP - Pause a task for a period of time`

Syntax

```
void TK_SLEEP(unsigned long ticks);
```

Parameters

ticks	Number of CTICKs to wait before being scheduled.

Description

Execution of the calling task is suspended for the specified number of system clock ticks (`CTICKs`). On InterNiche networking systems, clock ticks are tracked by the variable cticks, and the frequency is defined by `TPS` (ticks per second).

Tasks put to sleep with this call may be awakened before the indicated time by a call to `TK_RESUME()`.

Returns

Nothing

6.27 TK_SUSPEND

API Name

`TK_SUSPEND - Suspend execution of a task`

Syntax

```
void TK_SUSPEND(TASK tk);
```

Parameters

tk	Task ID of the task to be suspended. A task ID value of TK_THIS refers to the current task.

Description

When a task is suspended, the task flags are set to not ready. If the task being suspended is the current task, it is as if the current task called `TK_BLOCK()`. The task will not be run again until another task or interrupt handler calls `TK_RESUME()` with the suspended task's ID.

Returns

Nothing

6.28 TK_YIELD, tk_yield

API Name

`TK_YIELD () - Relinquish the CPU to another tasktk_yield() - Relinquish the CPU to another task`

Syntax

```
void TK_YIELD(void);
```

Parameters

None

Description

`TK_YIELD()` is called when the task code wants to wait for something to occur - a situation often referred to as a "busy wait". The `TK_YIELD()` primitive must give other tasks a chance to run, yet resume the calling task in a short interval. On a round-robin system like NicheTask this is easy - you simply mark to current task as runnable an call the round-robin scheduled.

On an RTOS where tasks have priorities, this can be somewhat trickier to implement. These systems sometimes support a call which will let tasks of equal or greater priority run, by not lower priority tasks. A task spinning on such a `TK_YIELD()` macro would never allow a lower priority task to run.

One remedy for this is to code the `TK_YIELD()` macro to put the task to sleep for a single clock tick. This will force it to wait a reasonable interval during which lower priority tasks may potentially get some cycles. The draw back is that even when the system has nothing else to do, the task spinning on will never be able to utilize all the CPUs power - it will always spend a certain amount of time gratuitously blocked.

Notes

The `tk_yield()` macro (same name in lower case) is identical to the uppercase version. It is supported for historical reasons.

Returns

Nothing

6.29 tk_sem_create

API Name

`tk_sem_create()`

Syntax

```
IN_SEM *tk_sem_create(int16_t maxcnt);
```

Parameters

maxcnt	maximum signal count

Description

Allocate and initialize a semaphore. The macro `SEM_ALLOC` creates a binary semaphore (max count = 1). However, the function `tk_sem_create()` will accept a larger `maxcnt`.

Returns

Pointer to newly created semaphore

Related Macro

`SEM_ALLOC()`

Macro Definition

```
#define SEM_ALLOC() tk_sem_create(1)
```

6.30 tk_stats

API Name

`tk_stats() - Show status for all threads`

Syntax

```
void tk_statsl(void *gio);
```

Parameters

`gio`	GIO to write output to.

Description

Write status of each task to the provided GIO, information on each task should include things like run state, stack usage, and anything else that may be interesting to system debug.

Returns

Nothing

6.31 tk_sem_free

API Name

`tk_sem_free()`

Syntax

```
void tk_sem_free(IN_SEM *sem);
```

Parameters

sem	pointer to semaphore to be freed

Description

Free a semaphore and wakeup any tasks listed in the semaphore's `tk_waitq`.

Returns

Nothing

Related Macro

`SEM_FREE()`

Macro Definition

```
#define SEM_FREE(s) tk_sem_free(s)
```

6.32 n_close

API Name

`n_close() - Accessor to shutdown network interface`

Syntax

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

Parameters

```
int if_number /* index into nets[ ] for NET to close */
```

Description

Does whatever is necessary to restore the device and its associated driver software prior to exiting the application. This function may not be required to do anything on embedded systems which start their devices at power up and don't have any reason to shut them down. If packet types (i.e.: `0x0800` for IP and `0x0806` for ARP) have been accessed in a lower layer driver, they should be released here.

Returns

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

6.33 n_init

API Name

```
n_init() - Accessor to startup network interface
```

Syntax

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

Parameters

```
int if_number /* interface number, for indexing nets[ ] */
```

Description

This routine is responsible for preparing the device to send and receive packets. It is called during system startup time after `prep_ifaces()` has been called, but before any of the other network interface's routines are invoked. When this routine returns, the device should be set up as follows:

- Net hardware ready to send and receive packets.
- All required fields of the net structure are filled in.
- Interface's MIB-II structure filled in as show below.
- IP addressing information should be set before this returns unless DHCP or BOOTP is to be used. See the section titled **"Initialization of net Structure IP Addressing Fields**".

This will usually include hardware operations such as initializing the device and enabling interrupts. It does not include setting protocol types. This is handled later (see the section **n_reg_type**. Upon returning from this routine it is safe for your hardware's interrupt or receive routines to start enqueuing received packets in the `rcvdq`. Packets which are not IP or ARP will be discarded by the stack.

The `nets[ ]` structure array element that is indexed by `if_number` should be completely filled in when this function returns. Note that the work of filling this structure is shared between `prep_ifaces()` and this function, so if all `nets[ ]` structure setup was done in `prep_ifaces()` (see **The 'glue' Layer**) there may be nothing to do here.

Shown below is an example of code that can be used for setting up the MIB structure for a 10 Mbps Ethernet interface. The `n_mib` field of the `nets[ ]` structure points to a structure that is used to contain the MIB information which has already been statically allocated by the calling code. See RFC1213 for detailed descriptions of the MIB fields. Most of the MIB fields are used only for debugging and statistical information, and are not critical unless your device is managed by SNMP. The `ifPhysAddress` field is an exception. It is used by ARP to obtain the hardware's MAC address and MUST be set up correctly for the IP stack to work over Ethernet. Note that although `ifPhysAddress` is a pointer, it does not point to valid memory when the MIB structure is created. The porting engineer should make sure it points to a static buffer containing the MAC address before this function returns. The size of this address is determined by the media (6 bytes for Ethernet) and should be set in the `nets[ ]` structure member `n_hal` (hardware address length).

```
u_char macaddress[6];   /* should contain interface's MAC address */

nets[if_number]->n_mib->ifDescr = "Ethernet Packet Driver";
nets[if_number]->n_mib->ifType = ETHERNET; /* SNMP Ethernet type */
nets[if_number]->n_mib->ifMtu = ET_MAXLEN;
nets[if_number]->n_mib->ifSpeed = 10000000; /* 10 megabits per second */
nets[if_number]->n_mib->ifAdminStatus = 1;
nets[if_number]->n_mib->ifOperStatus = 1;
nets[if_number]->n_mib->ifPhysAddress = ...macaddress[0]; /* example */
```

Returns

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

6.34 n_refill

API Name

`n_refill - Replenish device driver's internal resources.`

Syntax

```
void (*n_refill)(int iface);
```

Parameters

`iface`	interface number of device to refill

Description

Refills the device's internal packet buffer pool by calling `PK_ALLOC()` or `PK_CONTIG` to obtain packet buffers from the Stack's free packet buffer queues. The 'iface' parameter specifies the index of the device in the '`nets[iface]`' array. The `FREEQ_RESID` resource should be locked within the '`n_refill`' function prior to allocating the packets. The number of packets and their sizes is dependent upon the design of the driver. If there are multiple devices in the system, the developer can implement a single '`n_refill`' function for all of the devices or a separate '`n_refill`' function for each device.

The '`n_refill`' function is called in the `pktdemux()` function which is normally part of the main NicheStack task. The 'n_refill' function should no consider it an error if the device's internal packet buffer pool cannot be completely refilled.

Returns

Nothing

6.35 n_stats

API Name

```
n_stats() - Accessor to get network interface statistics
```

Syntax

```
int (*n_stats)(int iface, void *stats);
```

Parameters

```
int iface /* interface number to dump statistics for */
```

```
void * stats /* pointer to a user defined structure */
```

Description

OPTIONAL: `n_stats()` enables the driver to provide hardware specific information which is not included in the generic MIB-II interface group. This information might include hardware specific error counters, such as the number of collisions on an Ethernet link; or internal resource information, such as the status and number of current buffers available on a ring-buffer device. The definition of the '`stats`' structure and its contents is left to the driver writer. An example is the `enet_stats` structure in h`/ether.h`.

Returns

The function returns `ESUCCESS` if it is successful and `EFAILURE` if an error, such as a parameter value out of range, is encountered.

6.36 pkt_send

API Name

`pkt_send() - Insert frame into network driver queue`

Syntax

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

Parameters

```
typedef struct netbuf *PACKET
```

```
PACKET pkt /* pointer to netbuf structure containing frame to send */
```

Description

This routine is responsible for sending the data described by the passed `pkt` parameter and queuing the `pkt` parameter for later release by the device driver. If the MAC hardware is idle the actual transmission of the packet should be started by this routine, else it should be scheduled to be sent later (usually by an "end of transmit" interrupt (EOT) from the hardware).

The `PACKET` type is described in the section titled **The netbuf Structure and the Packet Queues**". All the information needed to send the packet is filled into the structure addressed by this type before this call is made. Some of the important fields are:

```
pkt->nb_prot;   /* pointer to data to send. */
pkt->nb_plen;   /* length of data to send */
pkt->net;      /* nets[ ]structure for posting statistics */
```

The data addressed by `pkt->nb_prot` may or may not have already been prefixed with a MAC layer header depending on how the `nets[ ]` structure associated with the interface (`pkt->net`) has been configured. The rule for determining whether the MAC layer header is present or not can be expressed with the following pseudocode fragment.

```
if ((pkt->net->n_mibifType == SLIP)
      || (pkt->net->n_mib->ifType == PPP)
      || (pkt->net->n_lnh== 0))
   the packet at pkt->nb_prot is not encapsulated with a MAC header;
else
   the packet at pkt->nb_prot is encapsulated with a MAC header;
```

If the if statement in the above pseudocode evaluates to `TRUE` then the packet at `nb_prot` is not encapsulated with a MAC header and it is up to the network interface code to transmit the MAC header that is appropriate for the network medium (if any). On the other hand, if the "if" statement evaluates to `FALSE` then appropriate MAC headers for media such as Ethernet or Token Ring will have been placed at the head of the buffer passed by the calling routine and are not the responsibility of this routine; however some drivers may have to access, strip or modify the MAC header if they are layered on top of complex lower layers. The ODI `pkt_send()` routine is an example of this (see `doslib/odi.c`).

Regardless of whether it is the responsibility of the network interface layer to transmit the MAC header, it is necessary for the network interface to transmit the `nb_plen` bytes starting at `nb_prot` plus "any" MAC header bias that was used to align the start of the IP header. For Ethernet devices, the macro `ETHHDR_BIAS` is sometimes defined to 2 bytes, to align the IP header at a 4 byte boundary. Likewise, the number of bytes to transmit in this case would be `(nb_plen - ETHHDR_BIAS)`, if `ETHHDR_BIAS` was defined to non-zero. When all the bytes are sent, the structure addressed by the `PACKET` type should be returned to the free queue by a call to `pk_free()`, which may be called at interrupt time. Do not free the packet before it has been entirely sent by the hardware, since it may be reused (and its buffer altered) by the IP stack.

The simplest way to implement this routine is to block (busy-wait) until the data is sent. This allows for fast prototyping of new drivers, but will generally hurt performance. The usual design followed by InterNiche in the example drivers is to put the packet in an awaiting_send queue, check to see if the hardware is idle, and then call a send_next_from_q routine to dequeue the packet at the head of the send queue and begin sending it. The "end of transmit" ISR (EOT) frees the just sent packet and again calls the send_next_from_q routine. By moving all the PACKETs through the awaiting_send queue we ensure that they are sent in FIFO order, which significantly improves TCP and application performance.

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

Slow devices (such as serial links), and hardware which DMAs data directly out of predefined memory areas, may copy the passed buffer into driver managed memory buffers, free the `PACKET` and return immediately; however they should be prepared to be called with more packets before transmission is complete.

Interface transmit routines should also maintain system statistics about packet transmissions. These are kept in the `IfMib` structure that is addressed by the `n_mib` field in each `nets[ ]` entry. Exact definitions of all these counters are available in RFC1213. At a minimum you should maintain packet byte and error counts since these can aid greatly with debugging your product during development and isolating configuration problems in field. Statistics keeping is best done at EOT time, but can be approximated in this call. The following fragment of code is a generic example:

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

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

Returns

Returns `0` if OK, else one of the `ENP_` codes. Since this routine may not be waiting for the packet transmission to complete, it is permissible to return a 0 if the packet has been successfully queued for send or the send is in progress. Error (non-zero) codes should only be returned if a distinct hardware (or lower layer) failure is detected. There is no mechanism to report errors detected in previous packets or during the EOT. Upper layers like TCP will retry the packet when it is not acknowledged.

See Also

raw_send

7 NicheStack Integration

Your driver is now ready for integration with the full NicheStack.

7.1 Final Steps

- Copy the whole port directory into your NicheStack repository under `ReferencePorts`.
- Integrate the `ipport.h` files, this involves creating an `ipport.h` in the new port directory of the NicheStack from the PORT portions of the harness version and the rest from the NicheStack version.
- If you are using make" (or gmake), cd to the port directory, and execute make. You may need to fixup the PORT (port.mk) and TOOLS (tool.mk) defines if you renamed the port while copying.
- If you use an IDE, you should create your project now using all of the non-ReferencePort sources, and the sources from the new port directory. Full details should appear in the NicheStack reference guide.

You may wish to return to the harness project if you find an issue with port specific portions while developing with the full stack as may help to isolate the issue.