Broadcom Home Networking Source Code Resource Guide

Status

Last modified: $Date: 2001/06/15 23:43:57 $

This document reflects the Release 2.34 code base.

Contents

Chapter 1: Getting Started

1.1 Getting The Embedded Source Code

Source Code License Agreement

A Source Code License Agreement (SCLA) is needed to obtain the source code for the BCM42xx driver. Once this agreement is in place, you can access to the latest source code available for the chipset as well as tools that will allow you to validate you design via Broadcom's Partner Web page at http://hnsupport.broadcom.com. The access code and password will be provided to you at that point.

Choosing the Right Driver Source

It is very important to decide beforehand, which version of the embedded source code you want to receive. Broadcom has sample source code for major run-time operating systems (RTOSs): VxWorks (BSD 4.3, 4.4 and END models), pSOS, Linux, and WinCE. If your design incorporates one of these RTOSs, then the choice for which code to obtain is easy. If, however, you use an alternate or proprietary RTOS, then the choice is a little bit more complex. In general, we tend to give the pSOS version of the code to customers because it has the simplest buffer architecture. But if you know that the buffer architecture of your RTOS is similar to one of the other ones, then perhaps pSOS is not the best choice. Please consult your contact at the Broadcom Home Networking Business Unit if you are unsure of which source code to obtain.

pSOS Caveats

Although pSOS has a simpler buffer architecture than the other versions of the code, it is lacking in its implementation of timer services. Basically, there is no way to register a callback function to be called after a specified timer interval has elapsed. Instead, for pSOS the timer functions are implemented in a less-than-optimal way. If your RTOS has proper timer services, please use those instead of the timer implementation in the pSOS source code. If you need assistance in this, please email hnsupport@broadcom.com with your questions.

1.2 BCM42xx Functionality Verification Checklist

The following are things that need special attention to during bring-ups. Please verify each of these before proceeding.

Big Endian/Little Endian

If your platform uses big endian byte ordering, you must set the flag IL_BIGENDIAN in the makefile (usually via "-DIL_BIGENDIAN" flag). If you fail to set this flag, the driver will not function properly.

DMA/Programmed I/O

The code can run in either programmed I/O mode (16-bit MSI) or DMA mode (if connected to a PCI bus.) You must make sure that the DMA compile flag is set properly. This flag is setup in the makefile usually via the compile option "-DDMA".

Slow Access Registers

There are two types of registers in the BCM42xx's register address space. They can be referred to as the "fast access registers" and "slow access registers." The bus timing for accesses to these registers is different. Please refer to the BCM42xx Datasheet for timing details. You must ensure that the timing for accesses to each type of register is met. This can be done in a number of different ways:

- **Processor Configuration:** Some processors allow for programming of different bus timings depending on the address range selected. This is the most cost-effective (both in terms of performance and component price) of the approaches and should be used if available. If you processor will not allow you to change bus timings for registers in the same device (chip select), then you may be able to dynamically adjust the bus timing when slow registers are going to be accessed. Since these registers are only accessed during initialization time, the timing could be slowed to meet the specifications of the slow access registers. Once the initialization code is

complete, the timing can be adjusted to the specification of the fast access registers.

- **External decode circuitry:** Implement external logic to decode the address of the register being accessed and adjust timing accordingly.
- **Software delays:** You can adjust the register access macros in the source code to behave differently if a slow register is being accessed. Slow access registers are latched into a holding register. You can use this fact to adjust the macros like this:
    - For the read macro, you can read it once expecting invalid data, delay the appropriate amount of time, and read that same register again. The second read will yield the correct data.
    - For the write macro, you must add a delay after the write operation to ensure that no successive write operation occurs before the specified time interval is complete.

General MSI Considerations

There are several considerations when using any of the MSI modes of the BCM42xx/BCM44xx. Please be sure to read the Application Note *BCM4210 MSI Mode Design Considerations* (part number 4210-AN702-R) before using any of the MSI interfaces. Read the **GenericMSIReadMe.txt** file (included on the CD) if you are using either Generic Asynchronous MSI or Generic Synchronous MSI bus interfaces on the BCM4210, BCM4211, or BCM4413 chip.

Chapter 2: InsideLine Driver Architecture

2.1 Introduction

This chapter provides an introduction to the internal design and implementation of the Broadcom BCM42xx/BCM44xx InsideLine™ Home Phoneline Networking device driver. The general structure and design methodology is presented along with a brief description of each *port* of the driver.

Since the device driver is evolving, the information presented here is subject to change without notice.

Copies of the HPNA protocol specification cannot be distributed by Broadcom. The spec is owned by the Home Phoneline Network Association. Your company MUST be a member to get a copy of the spec. See www.homepna.org for details on how to obtain this specification.

2.2 Background

InsideLine Technology

InsideLine technology includes a family of frame-oriented link, media access, and physical layer protocols that can scale over a large range of bandwidths on existing, arbitrary-topology, single-pair, residential telephone wiring using RJ-11 jacks.

- no hubs or central servers are needed
- 4-32 Mbps data rates, scalable to over 100 Mbps
- spectrum compatible: 4-10 MHz band allows concurrent POTS and U. ADSL
- QAM/FDQAM (frequency diverse QAM)
- per-packet equalizer training
- rate-adaptive constellation size (2-8 bits/baud @ 2 or 4 Mbaud)
- media access is simple broadcast CSMA/CD
- backwards compatible with HPNA 1.0 1-Mbps protocol
- LARQ - (Limited Automatic Retransmit reQuest) impulse error control protocol
- multicast is supported using Ethernet/IEEE 802 conventions

The Chip

The BCM42xx/BCM44xx InsideLine™ Controller (Block Diagram) is an integrated controller for a 16-Mbps data-rate InsideLine HPNA 2.0 network. The BCM42xx/BCM44xx provides a direct interface to the PCI bus, CardBus, or a Microprocessor Slave Interface for embedded systems. The chip supports 32-bit DMA and PIO (direct fifo) accesses in PCI/CardBus modes and 16-bit PIO access, exclusively, in MSI mode, with full packet buffering. A 4096-byte 16/32-bit PCI memory space is decoded at BAR0. PCI IO space accesses are not supported by the chip. The controller is PCI function #0 of all shipping chips. The digital chip currently interfaces with a separate, non-programmable, BCM41xx analog front end chip.

The term *DMA-mode* refers to use of the PCI master functionality of the chip, *PIO-mode* to refer to direct programmed I/O use of the chip transmit and receive FIFOs independent of the bus (PCI or MSI), and *MSI-mode* when referring specifically of the (PIO-only) Microprocessor Slave Interface functionality.

There are two DMA channels, one for transmit and one for receive. Each dma engine is configured via a set of four registers to process a contiguous vector (ring) of eight-byte descriptors. Descriptors are read-only - never written by the chip. Each eight-byte descriptor consists of a four-byte control word and a four-byte, 32-bit PCI address word. Each descriptor ring must be aligned on a 4096-byte memory boundary and, given the 4096-byte maximum contiguous dma segment size, each ring is limited to a maximum of 4096/8=512 entries. The chip

supports arbitrary byte alignment and length of data buffers with the restriction that buffers do not span 4096-byte memory boundaries.

There are two 16-bit direct fifo channels, one for transmit and one for receive. Each fifo is accessed via a control and a data register.

There is a pair of 32-bit IntStatus/IntMask registers that are used in DMA mode and a pair of 16-bit MSIIntStatus/MSIIntMask registers that are used in PIO (PCI/CardBus or MSI -- not restricted to MSI) mode.

There is a Device Control register (*devcontrol*) at address 0x0 which functions as a master switch to enable the modem, transmit and receive FIFOs, and various test functions.

There are many modem configuration registers which require only initialization-time access.

The chip includes two MACs:

- DFPQ (Distributed Fair Priority Queue) MAC is the native contention resolution algorithm used by InsideLine
- 802.3 BEB (Binary Exponential Backoff) MAC is included for backwards compatibility with the HPNA 1.0 protocol.

The chip includes two modems:

- InsideLine (HPNA 2.0) - 2-16 Mbps
- Tut (HPNA 1.0) - 1 Mbps

Three HPNA operating modes are supported.

iLine10 (aka *Native-mode*, aka *HPNA 2.0 mode*)
: This is the default, high-performance mode and is used when no HPNA 1.0 stations are detected. The DFPQ MAC is used.

Compatibility (aka *Mixed-mode*, aka *Gapped-mode*)
: If an HPNA 1.0 station is detected on the net, the driver dynamically puts the chip into compat or tut mode in order to interoperate with those HPNA 1.0 stations. This is a slightly modified version of native mode in which periodic gaps are inserted into the modulated HPNA 2.0 frame to ensure that HPNA 1.0 stations will detect carrier and defer. These gaps introduce a 10-20% performance impact. The Ethernet MAC is used.

HPNA 1.0 (aka *Tut mode*)
: 1 Mbps HPNA 1.0 modulation and framing. The Ethernet MAC is used.

All three frame types can be be received while in any operating mode. When in native mode, only native frames can be transmitted. When in compat mode

(mixed-mode), either gapped HPNA 2.0 frames or HPNA 1.0 frames can be transmitted. When in HPNA 1.0 mode, only HPNA 1.0 frames can be transmitted.

2.3 Chips Supported

The driver supports the following chips:

- 4210 (vendor = 0xfeda, device = 0xa0fa), "B0" (rev = 1) only
- 4230 is the cardbus version of the 4210 (vendor = 0xfeda, device = 0xa10e)
- 4211 (vendor = 0x14e4, device = 0x4211)
- 4413 (vendor = 0x14e4, device = 0x4410), rev >=1 only

The BCM4413 InsideLine core implements both 2 Mbaud and 4 Mbaud bands thus supporting both iLine10™ and iLine20™.

2.4 Ports List

The driver has been ported to the following operating systems and hardware platforms.

Name	OS	Hw Platform	DMA/PIO/MSI
NDIS	Win95	x86 PC	DMA/PIO
	Win98 and Win98SE	x86 PC	DMA/PIO
	NT4 (SP4)	x86 PC	DMA/PIO
	Win2000	x86 PC	DMA/PIO
	WinME	x86 PC	DMA/PIO
	WinXP	x86 PC	DMA/PIO (Beta 2 and above)
	CE v.2.12	BCM3310 MIPS	MSI
	CE v.2.12	x86 PC	DMA/PIO
	CE v.3.00	x86 PC	DMA/PIO
BSD	FreeBSD 3.x	x86 PC	DMA/PIO
	VxWorks 5.3.x	x86/MIPS	DMA/PIO/MSI
	VxWorks 5.4	x86/MIPS	DMA/PIO/MSI
	VxWorks END	x86/MIPS	DMA/PIO/MSI
Linux	2.2 kernel	x86/ARM/MIPS	DMA/PIO/MSI
	2.4 kernel	x86/ARM/MIPS/SPARC	DMA/PIO/MSI
pSOS	pSOS v2.5	x86 PC	DMA/PIO

Name	OS	Hw Platform	DMA/PIO/MSI
V2Hal*	pSOS v2.5	BCM93310 MIPS	MSI
	pSOS v2.5	BCM94100 MIPS	MSI
	2.20[bh]	BCM93350 MIPS	MSI
	2.20[bh]	BCM93352 MIPS	DMA

* Broadcom Cable Modems

2.5 Source Files

The driver is partitioned into OS-specific ("port-specific") and OS-independent ("common") source files. The naming convention is to use the **il**_ prefix for port-specific objects and **ilc**_ prefix for common objects.

The **Full** Source package includes all of the common files and all of the port-specific files for a particular OS. The **Basic** Source package does not include all of the common files listed below.

Common Files

ilc.[ch]
: Principle "common" code. Shared amongst all port-specific ports of the driver. Most ports of the driver should not require modifying ilc.[ch].

pe_select.[h]
pe_select_maxse.c
: Payload encoding (constellation) selection algorithms. The ".c" files are included in the Full source package only.

plarq.[ch]
: Portable Limited Automatic Retransmit Request Protocol implementation. These files are included in the Full source package only.

ilc_cert.c
: HPNA certification/diagnostic protocol support.

crc.[ch]
: crc32/16/8 routines

il_dbg.h
: Common header #includes the appropriate port-specific header file which defines tunables, printf/sprintf, bcopy/bcmp/bzero, register access macros, shared memory access macros, ASSERT, and trace/logging macros.

il_export.h
: Common header that defines the list of port-specific functions called by name from the common code. These functions must be provided by each port of the driver.

include/bcm42xx.h
: Chip-specific manifest constants and data structures.

include/bcmdevs.h

Broadcom Home Networking chip vendor and device id values.

include/bcmendian.h
Defines a few platform-specific big/little-endian macros and #includes the native endian.h

include/typedefs.h
Driver-specific common typedefs (uint32, uint16, etc).

include/bitfuncs.h
Processor/compiler-specific bit manipulation utility functions.

include/bcmdevs.h
Chip vendor and device identifiers.

proto/iline.h
InsideLine protocol-specific values.

proto/ilcp.h
InsideLine Control Protocol specific values.

proto/ethernet.h
Hacked version of the FreeBSD ethernet.h

NDIS Files

il_ndis.[ch]
main driver for NDIS 3.x/4.x/5.x

il_ndis30.[ch]
NDIS 3.0-specific shim functions

il_ndisCE.[ch]
CE-specific shim functions

ndis41.h
Unedited local copy of the 4.1 version of NDIS.H used instead of installed 4.0 version when compiling for NT4.

il_ddk.[ch]
DDK/WDM-specific low-level functions

il_ddkCE.[ch]
CE/DDK-specific low-level functions

il_stats.h
Stat counter API used exclusively by the NDIS driver.

include/epiioctl.h
NDIS custom OIDs used by the user-level **il** command

BSD Files

if_il.c
FreeBSD/VxWorks driver

il_bsd.h
FreeBSD/VxWorks specific header file.

include/ilsockio.h
BSD/Linux socket ioctls - used by the user-level **il** command

../VxWorks_pc/

VxWorks Intel PC platform BSP directory

Linux Files

il_linux.[ch]
Linux driver
include/ilsockio.h
socket ioctl definitions - used by the user-level **il** command

pSOS Files

il_psos.[ch]
pSOS driver
../pSOSv217_pc/
pSOS x86 BSP directory

2.6 Data Structures

il_info_t (port-specific)

This is the principle, port-specific, per-device-instance private structure. This per-port structure **must** begin with a common ilc_info_t because callbacks from ilc.c into the per-port routines cast their ilc_info_t* on the fly to the il_info_t . Typically, the il_info_t is tiny since it stores only OS- and hardware-platform-specific state.

Since this structure is port-specific, none of the common code (ilc.c, constel_select.c, etc) can #include it.

Most port-specific routines take a *il_info_t** as their first argument.
The il_info_t pointer local variable is usually named *il*.

ilc_info_t (ilc.h)

This is the principle, common per-device-instance data structure.
It is large and complex. Refer to the block comment and structure declaration in *ilc.h* for more details.

Most common routines take a *ilc_info_t** as their first argument.
The ilc_info_t pointer local variable is usually named *ilc*.

struct scb (ilc.h)

The driver maintains cached per-path state in a set of station control blocks hashed on the remote 6-byte MAC address. Typically there are 4 to 32 scbs statically compiled-in. There is one scb per active remote station or inuse

multicast/broadcast address. The scb is allocated and initialized on a lookup *miss* and reclaimed on-demand in LRU order. Information stored in the scb includes advertised rate descriptor, rate binary exponential backoff count and limit values, mixed-mode/tut-mode fallback state, constellation history (SNR/errors over the previous N received frames), and several per-path stat counters.

bcm42xxregs_t (bcm42xx.h)

Chip registers plus symbolic macros for all the register bits.

bcm42xxdd_t (bcm42xx.h)

DMA Transmit/Receive Ring descriptor structure.

bcm42xxrxhdr_t (bcm42xx.h)

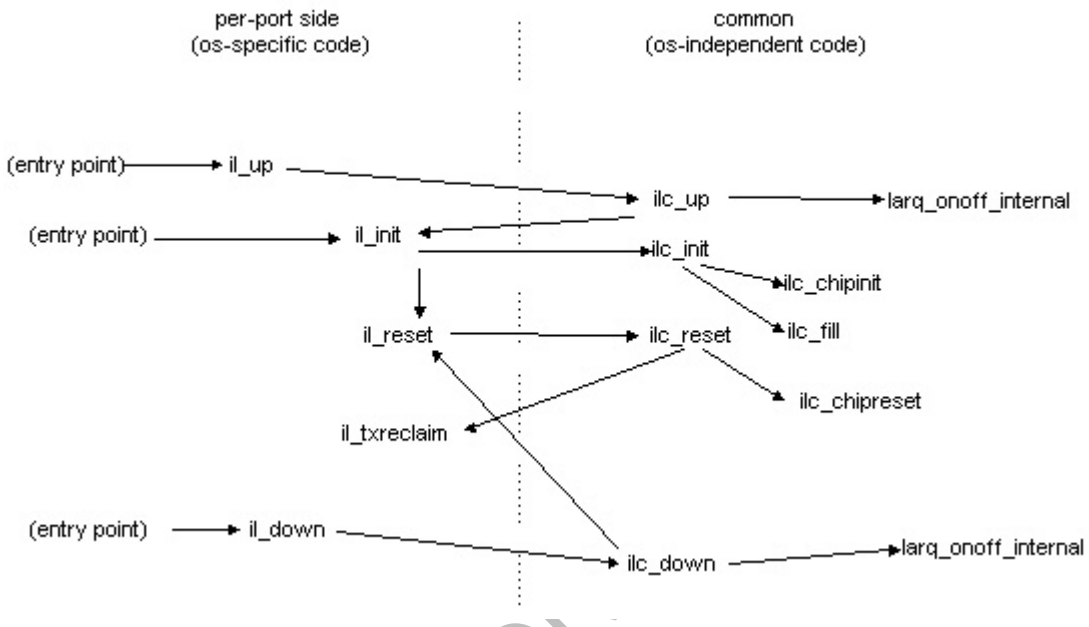
Received frames start with this 20-byte receive frame data header. There are two overlayed variations, bcm42xxrxhdr_t and bcm42xxv1hdr_t for received InsideLine and Tut frames, respectively.

In PIO mode, these are the first bytes read by the driver from the receive fifo. In DMA mode, this structure is placed (DMA written) by the chip at the start of the receive buffer. The received frame data (iline/tut frame header) starts at buf+HWRXOFF bytes into the receive buffer.

2.7 Function Call Graphs

In each of the function call graphs below, the lefthand side contains the OS-specific functions and the righthand side contains the OS-independent functions. For clarity, some of the lower-level functions are not shown.

up/down/init/reset



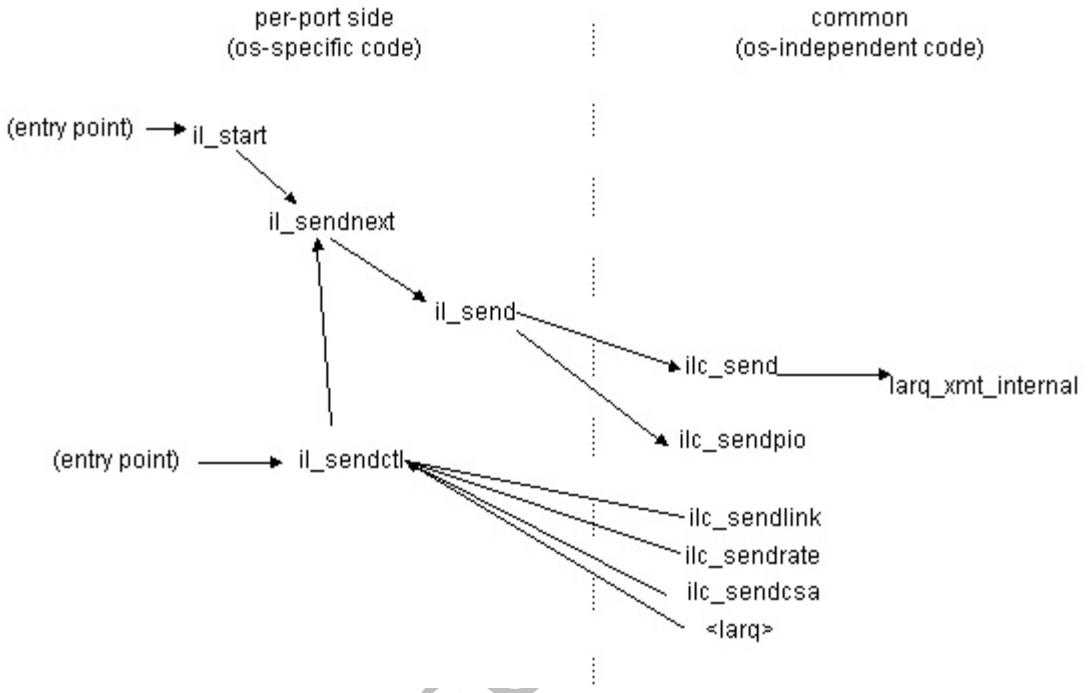
The driver state is logically comprised as several layers -- rings of an onion -- in the following outer-to-inner order:

- driver load/unload
- device probe+alloc+attach/detach/free
- up/down - be persistently operational/non-operational
- init/reset - transient sw/hw re-initialization
- chipinit/chipreset - transient hw re-initialization

il_reset and *ilc_reset* reset the logical sw/hw interface.
il_init and *ilc_init* reinitialize the logical sw/hw interface.
ilc_chipreset and *ilc_chipinit* reset and performs most of the chip-specific re-initialization, respectively.

Most mode changes and error handling is accomplished via a reinitialization (il_init). This "big hammer" approach simplifies the code and greatly minimizes the number of possible state interactions.

Transmit



This figure shows the basic transmit code path. Because most of the drivers support both DMA and PIO modes as a runtime selection, some packet buffer queue is required in addition to the underlying chip transmit ring or fifo. All driver entry points are port-specific. This figure is an example for the BSD driver (if_il.c). For comparision, the pSOS driver transmit entry point is *ni_send*.

The only exposed (non-static) functions are the driver transmit entry point (*il_start* in the case of the BSD and Linux drivers) and the send control function *il_sendctl* which is used by the several common *ilc_sendxxx* routines and LARQ.

There is always an exported "start" entry point, sometimes a private "sendnext" intermediate function, and always an exported "send" hardware-transmit function.

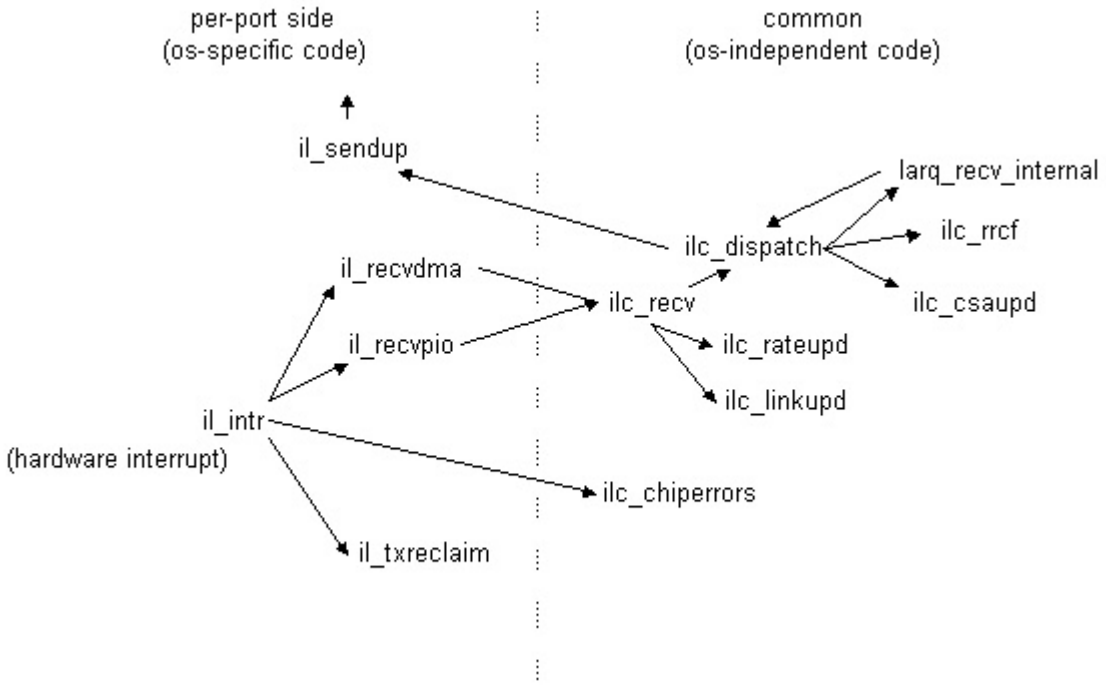
Various routines enqueue their packet buffers and call il_sendnext() to transmit the frame. il_sendnext() determines if the internal resources required to transmit a frame are available. If they are, it calls il_send(). If they are not, it just returns. In PIO mode, the pointer *pioactive* indicates whether a transmit packet is already pending to the chip. In DMA mode, pioactive will always be null. PIO-mode tx reclaim means just freeing the packet pointed to by *pioactive*.

ilc_send is the common send "helper" function which does most of the non-chip-specific transmit functionality including calling larq_xmt_internal() which may prepend a LARQ protocol header. There is a surprising amount of scb lookup, link-layer control, priority, and queueing complexity hidden in ilc_send().

The Linux driver, for example, processes sk_buffs which are single contiguous network buffers. The BSD driver processes chains of mbufs. None of these port-

specific packet buffering structures can be accessed directly by the common code. Rather, the generic packet primitives (il_pget, il_ppush, etc) are used.

Interrupt Handling



This figure shows the basic interrupt code path. The port-specific *il_intr* interrupt handler is a driver entry point called by the operating system. It identifies the particular interrupt event (tx, rx, error) and mode (dma, pio) and calls the appropriate port-specific or common function to process the event.

In PIO mode, transmit interrupts must be enabled. In DMA mode, transmit interrupts can be enabled or can be dynamically enabled or disabled, depending on port-specific buffering issues. For example, in DMA mode, the BSD, Linux, and pSOS drivers run with transmit interrupts disabled until the tx dma ring fills and the first packet is left on the software tx queue by il_sendnext(). At this point transmit interrupts are enabled and remain enabled until the software tx queue is empty. The NDIS driver requires a more complex txreclaim scheme and so runs with transmit interrupts always enabled.

The il_txreclaim() routine reclaims any completed packets and starts the next one.

Either il_recvdma() or il_recvpio() is called to process received frames from the chip and call the common ilc_recv() helper routine, which does all the non-chip-specific work. Of which, again, there is a surprisingly large amount.

ilc_recv() is responsible for all of the software receive-side per-*frame* processing. ilc_recv() calls ilc_linkupd() to maintain the software link integrity state and ilc_rateupd to determine the optimal band and constellation to advertise to the remote station. If the received frame is an ILCP packet (ether_type 0x886c), then ilc_dispatch() is called to the process the ILCP header, otherwise ilc_recv() calls the per-port function il_sendup() to prep and pass the packet buffer up to the higher layer protocols.

ilc_dispatch() processes all ILCP packets. Since ILCP headers may nest, ilc_dispatch() is implemented as a loop which, at each iteration, identifies the next ILCP ether_type and calls the appropriate routine to process it. When the next_ethertype is null, the loop terminates and il_sendup() is called to pass any remaining bytes up to the higher layer protocols.

ilc_dispatch() calls larq_recv_internal() to process received LARQ frames. LARQ frames may (1) have their header stripped and be subject to continued processing by ilc_dispatch, (2) be completely consumed by larq_recv_internal(), or (3) be buffered by larq_recv_internal() and later re-injected back into ilc_dispatch().

Generic Packet Primitives

The per-port portion of the driver must provide the following set of packet primitives to allow common code to allocate, adjust, dup, and free native network packet buffers.

il_pget
: Allocate a native packet buffer of a given size. Returns a void* handle to the packet structure and, optionally, starting virtual and physical addresses of the data associated with the packet.

il_pfree
: Free a packet buffer specified by the void* handle.

il_ppush
: Add n bytes to the front of a packet buffer. Subtracts the specified number of bytes from the native buffer pointer and adds this value to the native buffer length.

il_ppull
: Remove n bytes from the front of a packet buffer. Adds the specified number of bytes to the native buffer pointer and subtracts this value from the buffer length.

il_plen
: Return the total number of bytes in the packet buffer.

il_psetlen
: Set the total number of bytes in the packet buffer.

il_pdup

Logically duplicate a packet buffer and return the void* handle of the new packet buffer. The current implementation does this via a data copy operation and the resulting new packet buffer is readable and writeable. This might change in future versions of the driver.

Debug, Trace, and Log Primitives

The per-port portion of the driver must provide the following set of primitives to allow common code to support debugging and tracing operations. These functions are used when DBG or BCMDBG are defined.

il_assert
: Cause an exception when the condition is not true.

il_log
: Calculate the cycle and instruction count deltas from the previous il_log call. Call ilc_log with the counts and the passed in parameters.

Portable LARQ Timer Support

The per-port portion of the driver must provide the following set of timer primitives to allow the OS-independent implementation of the LARQ protocol to schedule asynchronous events:

il_larq_add_timer
: Schedule a pending LARQ timer. Timer resolution of 1 millisecond is desireable although 10 millisecond resolution is acceptable.

il_larq_del_timer
: Cancel any pending LARQ timer.

In addition, the per-port portion of the driver must define and appropriately register a function, typically named il_larq_timer, to be called when the LARQ timer fires. This function should acquire the perimeter lock, call the common code function *larq_timer_internal*, walk the tx queue, and drop the perimeter lock. (See Locking Model for information on the perimeter lock.)

2.8 il/ilc API

The driver is partitioned into port-specific and common source files. The interface between the port-specific and the common code in ilc.c deserves some description because it is this interface that gets ported each time.

ilc.c Exported Routines

Prototypes for the exported ilc functions are declared at the bottom of ilc.h. With rare exception, all ilc functions take a ilc_info_t* as their first argument.

Port-Specific Exported Routines

Common code cannot reference il_info_t since this data structure is port-specific. The common ilc.c code makes some callbacks into the port-specific driver.

Those port-specific functions that are exported to the common code are declared in il_export.h:

```
        /* misc callbacks */
        extern void il_init(void *ilh);
        extern void il_reset(void *ilh);
        extern uint il_sendctl(void *ilh, void *p, int txpri, int
txbpb);
        extern void il_txreclaim(void *ilh, bool forceall);
        extern void il_rxreclaim(void *ilh);
        extern void il_sendup(void *ilh, void *p, uint pri);
        extern uint il_ms(void);
        extern void il_delay(uint us);
        extern void il_link_up(void *ilh);
        extern void il_link_down(void *ilh);
        extern int il_up(void *ilh);
        extern int il_down(void *ilh, int reset);
        extern void il_dump(void *ilh, uchar *buf, uint len);
        extern void *il_malloc(uint size);
        extern void il_mfree(void *addr, uint size);

        /* canonical packet primitives */
        extern void *il_pget(void *ilh, bool send, uint len, uchar **va,
uchar **pa);
        extern void il_pfree(void *ilh, void *p, bool send);
        extern uchar *il_ppush(void *ilh, void *p, uint bytes);
        extern uchar *il_ppull(void *ilh, void *p, uint bytes);
        extern uint il_plen(void *p);
        extern void il_psetlen(void *p, uint len);
        extern void* il_pdup(void *ilh, void *p);

        /* larq timer primitives */
        extern void il_larq_add_timer(void *ilh, uint ms);
        extern int il_larq_del_timer(void *ilh);
```


2.9 Locking Model

Although the specifics vary, all ports share the following approach towards restricting thread preemption and concurrent execution:

- All data structures are enclosed within a single perimeter.
- This perimeter is enforced either (1) transparently by the operating system, (2) explicitly by the driver via a mutual-exclusion lock, or (3) a combination of the two.

- The perimeter lock is acquired and released by the port-specific entry points.
- The details of the perimeter lock are port-specific, therefore any state associated with the perimeter lock is maintained in the il_info_t data structure.
- The perimeter lock must be acquired before any common code is called (common code expects the perimeter lock to be held across the call).
- Common code never explicitly references the port-specific perimeter lock.
- The perimeter lock is *not* acquired recursively.

2.10 Port-specific Design Notes

The BSD, Linux, and pSOS drivers are more similar than different and almost identical at the functional level. All three are relatively small and simple compared to the NDIS driver or to the common ilc.c code.

NDIS

For several reasons, the NDIS il driver is considerably larger and more complex than the other driver ports.

The NDIS il driver supports NDIS3 (Win95, WinCE), NDIS4 (NT4), and NDIS5 (Win98, Win98SE, Win2000, Millennium). The NDIS and DDK/WDM variations are encapsulated either within #ifdef WDM within the principle il_ndis.c file or within separate small source files (il_ndis30.[ch], il_ndisCE.[ch]). A small amount of necessary non-NDIS functionality is implemented within il_ddk.[ch] and il_ddkCE.c.

Two spinlocks are used - a general perimeter "lock" and a dpc-specific "dpclock."

There are two transmit NDIS packet queues, *txq* and *txdone*. All NDIS packets arriving at the MiniportSendPacket(s) entry point (il_msendpackets) are enqueued on txq. There is also a queue of local buffers *txctlq*, where the control packets (generated locally in the driver) are queued. The process of sending packets is accomplished in *il_sendnext*. *txctlq* is processed first, after which *txq* is handled. The txq queue has no flow control mechanism - it is of infinite size. Once the packet has been successfully or unsuccessfully transmitted, it is placed on the txdone queue and the routine il_sendcomplete() called to NdisMSendComplete() all completed packets. Great pains are taken to meet the constraint that NdisMSendComplete() be called only from a Dpc/Timer routine with no private locks held.

There are two lists of free local buffers(*lbuf*s), *txfree* and *rxfree*. At Miniport initialization, shared memory pages are allocated and split into one or more 2-Kbyte shared memory buffers to create each free list of local buffers. For transmit, the data comprising each original ndis packet chain is copied into a

single, contiguous 2-Kbyte local tx buffer and the necessary LARQ and HPNA 1.0/2.0 headers added. For receive, the packet data is DMA'd directly into a single, contiguous 2-Kbyte local rx buffer, HPNA and optional ILCP headers removed and processed, and the packet sent up to the higher level protocols.

The NDIS driver supports several features not provided in other ports of the driver:

Power Management (PM)
: NDIS-specific power management OIDs, programming the chip wakeup filter entries, entering and leaving D3 cold state.

Branding and Vendor-Specific Registry Issues
: NDIS driver customers are more concerned with driver branding and providing vendor-specific options such as changing the default values for link level procotols and modes based on registry values, and querying arbitrary OEM data contained within the hardware serial prom.

il_stats.h based stat counter API
: Interface into the driver ilc_info_t stat counters.

WinCE

WinCE implements a variant of the NDIS3.0 interface. The Windows CE variations to the NDIS driver are contained within the source files il_ndisCE.[ch], il_ddkCE.c and the inline conditional #ifdef UNDER_CE.

BSD

The BSD driver, *if_il.c*, currently supports FreeBSD 3.x, VxWorks 5.3, and VxWorks 5.4 END driver models. All three are implemented as small wrappers around a single basic BSD device driver and share the majority of code.

Transmit mbuf chains are enqueued on the if_snd queue and drivers il_start() routine called which could be renamed il_sendnext() since it is serving in this capacity. There are no private buffers allocated nor maintained in the driver. DMA reads and writes are done directly from/into the transmit and receive mbufs, respectively.

The silly differences in BSD mbuf variations are isolated nicely by a couple of macros and a pair of internal il_mget() routines, one for FreeBSD 3.x and one for the older simpler version of VxWorks mbufs.

Because the version of VxWorks supported is based on an older BSD code base, arguments are typically integer "unit" numbers instead of struct ifnet "*ifp" pointers. The VxWorks-specific wrappers which take a unit arg have names starting with "il_vx".

The source file *ilsockio.h* defines several iline driver ioctl command values used by the BSD, Linux, and pSOS drivers.

The perimeter thread exclusion is implemented via splimp()/splx().

Two modes of the BPF (Berkeley Packet Filter) are supported in FreeBSD. Ethernet (DLT_EN10MB) mode is the default. "Extended" (DLT_ILINE) is enabled via the SIOCSILBPFEXT ioctl and sends up the entire raw received frame including chip-specific rxhdr_t, pad bytes, HPNA 1.0 or 2.0 header, and Ethernet frame.

Linux

il_linux.c currently supports the 2.2 and 2.4 linux kernels, MODULE loading, and the PCMCIA module CardBus interface. The CardBus implementation must be compiled as a loadable module and, when used with the 2.2 kernel, requires the pcmcia-cs-3.1.22 (or newer) package. Also, BCM4230 chip registers at offsets 0x40-0x7C and 0x400-0x415 cannot be accessed properly when using the CardBus implementation with the 2.2 kernel.

No private buffer pools are needed. il_start() is the transmit entry point and enqueues the skb onto the private txq . il_sendnext() determines if the tx ring has available slots in DMA mode, or pioactive is NULL in PIO mode, before dequeueing the skb from the private txq . il_sendnext() also monitors the software txq to dynamically enable/disable tx interrupts when necessary and to set and clear the tbusy flag which flow-controls the upper protocol layers. Since skbs do not chain, il_send() is almost trivial.

Received frames are DMA'd directly into allocated sk_buffs or are PIO copied directly from the receive fifo register into an sk_buff before the common ilc_recv() is called.

A single spinlock (spin_lock_irqsave()/spin_unlock_irqrestore()) is used to singlethread the majority of the driver.

Some linux kernel networking code allocates skb's and reserves headroom based on the value of dev->hard_header_len . Since the il driver must prefix packets with up to 12 bytes of InsideLine and LARQ protocol header, we bump up our dev->hard_header_len by this amount to ensure sufficient skb headroom in il_send(). This means we must have private versions of eth_header() and eth_rebuild_header() that add 12 bytes of pad which means we must also add this padding in il_sendctl() and strip it off in il_send() . Also, since the kernel hardware header cache supports header lengths only up to 16bytes, the il driver can't use the hard_header cache.

The Linux UDP implementation implements driver transmit-side flow-control. At most 'sockbuf' number of frames are allowed to be pending for transmit at any one time. This requires the linux il driver run with transmit interrupts enabled and not be too lazy in reclaiming completed transmit packets.

pSOS

The pSOS il driver, il_psos.c, is a PNA+ NI driver and has been tested with pSOS 2.5 on an Intel x86 platform and several MIPS-based Broadcom hardware platforms.

The PNA+ packet buffer data structure is based on STREAMS (mblk_t, dblk_t, data triplets). Unfortunately the mblk allocator provided on some platforms allocates data buffers on arbitrary word alignments and which cross 4-Kbyte boundaries. This presents a problem since our chip cannot DMA address buffers which cross a 4-Kbyte boundary. For transmit, in DMA mode, the driver resolves this issue by detecting mblk data buffers which cross 4-Kbyte boundaries and splitting these into two tx descriptors. For receive, the driver statically allocates the maximum necessary number of 2-Kbyte receive buffers and esballoc() and "loans" these up to the higher protocol layers using esballoc().

The pSOS il driver has no access to a heap memory allocator so it must statically allocate necessary receive buffers and a few Kbytes of il_malloc/il_free buffering as part of its il_info_t structure.

pSOS doesn't seem to provide an integrated timer facility, therefore the il driver implements the periodic one second watchdog timer and the non-periodic LARQ timer via two dedicated kernel tasks. Developers using the pSOS port as a reference should only use the pSOS LARQ timer code if their target OS does not support a native timer object.

2.11 il dump Example

You are not required to implement a debugging "dump" routine but it is very often useful.
Here is an annotated output from the Linux "il dump":

The first paragraph comes from the port-specific il_dump():

```
il0: May  8 2000 18:32:31 version 2000.4.9.0
il c6b8c000 dev c798c6c0 name eth1 tbusy 0 txq.qlen 0
```

The next paragraph is the guts of all the software state of ilc_info_t.

```
ilc 0xc6b8c000 regs 0xc8834000 msglevel 1 debugstr ""
```

```
        up 1 promisc 0 promisctype 0 loopbk 0 linkint 1
        txbpb -1 txpri -1 enic2inic 0 piomode 0 pioactive 0x0
        larqlevel 1 larq_handle 0xc6b84000 nmulticast 1 allmulti 0
        vendor 0x14e4 device 0x4211 chiprev 0
        subvendor 0x14e4 subid 0x103 boardrev 1.0 afe 3
        perm_etheraddr 0:90:4c:9:0:6 cur_etheraddr 0:90:4c:9:0:6
        txd 0xc6b8b000 txdpa 0x6b8b000 txp 0xc6b8c154 txin 30 txout 31
        rxd 0xc6b8a000 rxdpa 0x6b8a000 rxp 0xc6b8c564 rxin 129 rxout 145
        macmode 2 v1_detected 0 v1_signaled 0 configmode 0 localforce 0
forcetut 0
        pcom 1 ctr1linkrcvd 0 xmt_tut 0 xmt_gapped 0
        intmask 0x1ff80 dogtimerset 0 linkstate 2
        csa 1 csa_rtx 0 csa_configmode 0x0
        csa_newtx 0x81000002 csa_prevtx 0x81000002 csa_oldtx 0x81000002
        csa_newrx 0x81000002 csa_prevrx 0x81000002
```

We're "up". Link integrity is "on." We've been compiled with -DDMA so support DMA exclusively. A single discrete multicast address is enabled. This is a rev0 bcm4211 chip. The interface is operating in native iLine10 (macmode 2) mode with no HPNA 1.0 stations detected directly (v1_detected) or indirectly (v1_signaled) and no local nor csa mode forcing in effect. csa is enabled and we're advertising only priorities 0 and 7 inuse and native mode.

Next paragraph is a dump of the software stat counters (aggregate since boot).

```
        txframe 71455 txbyte 10681872 txerror 0 rxframe 70017 rxbyte
4948143 rxerror 154
        txctl 65224 txcoll 76526 txcollframes 51030 txexcol 0
        txnobuf 0 txuflo 0 txherr 0 dmade 0 dmada 0 dmape 0 reset 5
        rxctl 63791 rxrterr 0 rxhcrc 0 rxpcrc 0
        rxcsgap 0 rxcsoverrun 0 rxsat 0 rxtrk 0
        rxrunt 18 rxgiant 66 rxotherr 0 rxdmauflo 0 rxoflo 0
        rxnobuf 0 rxfiltered 0 rxmiss 0 rxdemoderr 70
        rxfterr 0 rxifgv 0 rxsigv 0 rxstackbub 0
```

Another view of the priority mapping tables in use:

```
        txpri map:      6 5 5 6 6 6 7 7
        txllpri:        6231 0 0 0 0 0 0 65224
        txdfpqpri:      0 0 0 0 0 0 6231 65224
```

Here are all the chip memory space registers:

```
        devcontrol 0xe8 devstatus 0x200 compat 0x7 wakeuplength
0x80808080
        intstatus 0x1000000 intmask 0x1ff80
        afestatus 0x3 afecontrol 0x0
        frametrim 0x74a3 gapcontrol 0xa9d2 trackerdebug 0xa5
        festatus 0x382 fecontrol 0x0 fecsenseon 0xc24 fecsenseoff 0x2326
cspower 0x30
```

```
        modcontrol 0x0 moddebug 0x8000 cdcontrol 0x2406 noisethreshold
0xff03
        framestatus 0x0 framecontrol 0x697042
        enetaddrlower 0x4c090006 enetaddrupper 0x90
        maccontrol 0xff pcom 0x1
        gpiooutput 0x0 gpioouten 0x0 gpioinput 0xff
        msiintstatus 0x10 msiintmask 0x27
        funcevnt 0x8000 funcevntmask 0x0 funcstate 0x2
        intrecvlazy 0x1000000
        xmtcontrol 0x1 xmtaddr 0x6b8b000 xmtptr 0xf8 xmtstatus 0x20f8
        rcvcontrol 0x3d rcvaddr 0x6b8a000 rcvptr 0x488 rcvstatus 0x1408
        ctrcollisions 0x1209 ctrcollframes 0xc0d ctrshortframes 0x9
ctrlongframes 0x18
        ctrfilteredframes 0x0 ctrmissedframes 0x0
        ctrerrorframes 0x1a ctr1linkrcvd 0x0 ctrrcvtooshort 0x0
ctrxmtlinkintegrity 0x0
        ctr1latecoll 0x0 ctrifgviolation 0x0 ctrsignalviolation 0x0
ctrstackbubble 0x0
        qosusage[0] 0x0 qosusage[1] 0x0 qosusage[2] 0x0 qosusage[3] 0x0
        qosusage[4] 0x0 qosusage[5] 0x0 qosusage[6] 0xa4b0e qosusage[7]
0x64cbb7
        qoscoll[0] 0xb qoscoll[1] 0x0 qoscoll[2] 0x0 qoscoll[3] 0x0
        qoscoll[4] 0x0 qoscoll[5] 0x0 qoscoll[6] 0x1204 qoscoll[7] 0x2
        qosmacidle 0xebf34e67
```

And here are the Tut (HPNA 1.0) specific registers:

```
        tutctl 0x2140 tutaidcw 0x4406 tutaids 0x3 tutdt 0x29f4
        tutmc 0x0 tuttdcst 0x198 tuttaidcst 0x342 tutb 0x23f
        tutds 0x10 tutsid 0x21 tutsctl 0x1df tutnctl 0xe03
        tutifsc1 0x1de tutifsc2 0xed tuts1c 0xeee tutsc 0x2555
        tutdcst 0x198 tutaidcst 0x342 tuttgdcst 0x139 tuttgaidcst 0x2e3
        tutgaids 0xf03 tutgms 0xe04 tutss 0x3003
```

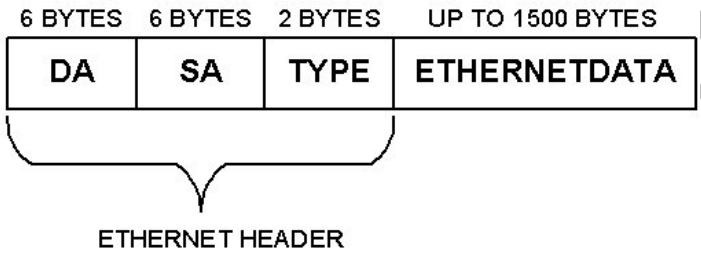
2.12 On-the-Wire Frame Format

An InsideLine (HPNA 2.0) frame is a bytestream consisting of a four-byte HPNA 2.0 "framecontrol" header followed by a payload, which is normally a standard Ethernet frame, followed by the standard Ethernet four-byte CRC32 frame check sequence, followed by an additional two-byte CRC16. The minimum frame size is 4+64+2=70 bytes. The maximum frame size is 4+1526+2=1532 bytes. Refer to the *iline.h* and *ilcp.h* header files and HPNA 2.0 protocol specification for more complete details.

An HPNA 1.0 (Tut) frame is a bytestream consisting of a four-byte pcom field followed by a payload that is normally a standard Ethernet frame.

2.13 Building InsideLine Frames for Transmission

The port-specific transmit entry point is called with a chain of one or more linked native packet buffer structures containing a standard Ethernet frame consisting of a 14-byte Ethernet header and up to 1500 bytes of user data. The four-byte Ethernet CRC32 is not present at this time since it is generated and appended to the frame by the hardware when transmitting the frame.

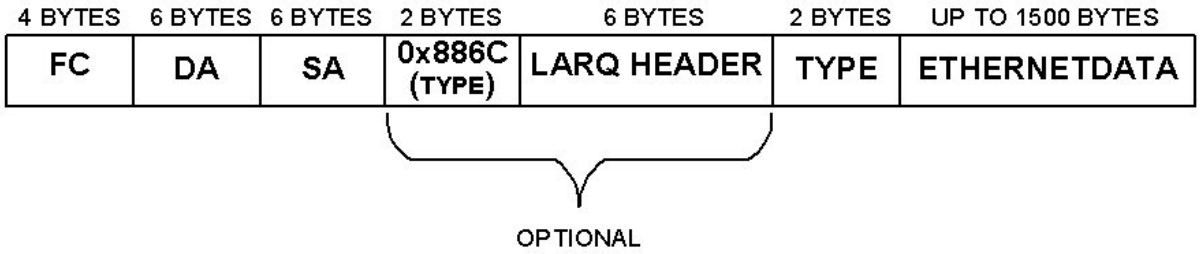


This native packet buffer is tagged by the driver with two bytes of InsideLine transmit priority (txpri) and transmit payload encoding (txpe) before being enqueued on the driver private tx queue. Depending on the native OS, these two bytes of information is either written into the packet buffer meta structure (mbuf, mblk, skb, whatever) or prepended onto the front of the packet buffer.

In DMA mode (when transmit descriptors are available) or PIO mode (when the chip transmit fifo is free) the packet buffer is dequeued, txpri and txpe are retrieved from the packet buffer and the per-port il_send() function is called to transmit the frame.

il_send() ensures that there are at least 12 bytes (TXOFF) of free space in the first buffer of the chain and that the 14-byte Ethernet header is contiguous and immediately following this. il_send() then calls ilc_send() which, among other things, always adds either a four-byte InsideLine framecontrol header or a four-byte HPNA 1.0 (Tut) header to the packet buffer and optionally, if larq is enabled and the station is in native InsideLine mode, calls larq_xmt_internal() to add an eight-byte ILCP/LARQ header to the packet buffer. The ILCP/LARQ header is inserted by larq_xmt_internal() between the Ethernet header Source Address field and the Ethernet Type field.

When ilc_send() returns to il_send(), the packet buffer will have the following format:



il_send() then posts this packet data to the chip for transmit indicating that it is either an InsideLine frame or a HPNA1.0 frame.

For an InsideLine frame, hardware calculates and updates the CRC8 field of the framecontrol header, calculates and appends both a four-byte CRC32 and additional two-byte CRC16 FCS fields, and pads the frame out to the minimum frame length (70 bytes). This is the on-the-wire frame described above that is transmitted on the media.

For an HPNA1.0 frame, hardware calculates and appends the four-byte CRC32 and pads the frame out to the minimum frame length (68 bytes).

2.14 Mode Switching and Fallback

The driver dynamically switches between native, mixed-mode, and tut mode if HPNA 1.0 stations are detected on the network. In native mode, if a HPNA 1.0 or mixed-mode frame is detected, the driver transitions into HPNA 1.0 mode for a period of two minutes. At the end of this period, it tries to transition into mixed-mode. While in mixed-mode, if no HPNA 1.0 stations are detected for a period of two minutes, the driver reverts to native mode. The section below provides some detail on this complicated topic that you can skip over.

HPNA 2.0 stations interoperate with HPNA 1.0 stations, as well as with other HPNA 2.0 stations in the presence of HPNA 1.0 stations.

Approaches

To achieve the above objectives, HPNA 2.0 stations are able to send and receive HPNA 1.0 format frames. However, in the presence of HPNA 1.0 stations, HPNA 2.0 stations should not send native HPNA 2.0 format frames to one another, since the HPNA 1.0 stations MAC will fail to recognize these frames, and thus the network's MAC will fail overall. Instead, HPNA 2.0 stations have the option of sending either HPNA 1.0 format or HPNA 2.0-"gapped" format frames to one another.

Under most conditions, it is preferable for HPNA 2.0 stations to send gapped frames, since high throughput is still possible, with only 10-20% performance loss from the native HPNA 2.0 format. However, in cases of high line attenuation, HPNA 2.0 stations cannot detect gapped frames even though they might still be able to demodulate them, and even though they can still detect HPNA 1.0 format frames. Under these conditions, it is preferable for HPNA 2.0 stations to send HPNA 1.0 format frames.

Another case in which HPNA 2.0 stations should send HPNA 1.0 format frames is when HPNA 1.0 stations cannot detect gapped format frames. Unfortunately, there is no straightforward method for an HPNA 2.0 station to infer this about an

HPNA 1.0 station, and the HPNA 1.0 station cannot explicitly notify the HPNA 2.0 station of this case.

Mode Switching and Fallback Overview

Mode switching refers to the situation when an HPNA 2.0 station either detects an HPNA 1.0 station -- indicating that only HPNA 1.0 or gapped format frames may be sent -- or concludes that no HPNA 1.0 stations remain "active" and native format frames may be sent once again.

Fallback refers to the situation when an HPNA 2.0 station, in the presence of HPNA 1.0 stations, decides to "fall back" to HPNA 1.0 format frames when it cannot detect gapped frames.

A more appropriate name is "format selection," as HPNA 1.0 / gapped format selection is a logical extension of rate selection. Much like rate selection, format selection is split into different tasks for data receivers and data senders. Also, like rate selection, format selection requires HPNA 2.0 stations to maintain seperate state information on "link quality" associated with each additional HPNA 2.0 station. Finally, like rate selection, changes in format selection are communicated by data receiver to data sender using the same backed-off rate request message protocol used in conventional rate selection. Note that there is no need for format selection when communicating with HPNA 1.0 stations, in which case HPNA 1.0 format is always used.

In the process of developing a format selection technique, two other fallback modes were implemented as options for use in case of "last resort.":

1. Always send gapped format in the presence of HPNA 1.0 stations
2. Always send HPNA 1.0 format in the presence of HPNA 1.0 stations

These modes are self-explanantory and are not discussed herein.

Per Station state vs Global state

In network of N+1 stations, a given HPNA 2.0 station, "Node A," simulataneously maintains N Data Sender state machines (one for each destination D_i, i = 1...N), and N Data Receiver state machines (one for each source S_i, i = 1...N). These state machines are distinct, since some of the link quality measurements vary by source / destination pairs. However, as is described later, there are several global conditions that affect all of Node A’s Sender or Receiver state machines at once:

- Changes in (V1_DETECTED or V1_SIGNALED) affect all of Node A’s Sender and Receiver state machines.

- Any destination indicating a desire to receive HPNA 1.0 format frames affects all of Node A's Sender state machines.
- Changes in the HPNA 1.0 receiver squelch value affect all of Node A's Receiver state machines.

2.15 Other Miscellaneous Functionality

Probe/Attach/Allocation

The mechanics of driver registration, loading, and probing, tend to vary from port to port. Both the BSD (FreeBSD 3.x) and Linux drivers support a loadable module interface as well as the option for being compiled and linked directly into the kernel.

After the device has been detected, the sequence of operations tends to be similar:

- Allocate il_info_t memory and bind this to the ifnet/device structure.
- Map chip registers.
- Confirm that the chip is one of ours.
- Read the chip revision.
- Reset the chip.
- Initialize spinlock or transmit queues.
- Register our interrupt handler.
- If in DMA mode, allocate tx and rx descriptor rings.
- Call *ilc_attach* which checks that the BCM42xx chip is accessible and performs some common attach-time initialization.
- Allocate and init the grotty stats data structure.
- Optionally, set/override the default local Ethernet address.
- If_attach/enable our entry points.

If any of these operations fail or we support unloading the driver, the *il_free* routine gracefully deallocates all of this.

Conditional Compile Flags

The following driver-specific, os-independent compile-time flags are used:

IL_PROTOS
: Encapsulates most InsideLine Link Level protocols including CSA, LARQ, Constellation selection, Rate Negotiation, and Fallback algorithms. Link and HPNA 2.0 Cert protocols are not included in this conditional.

IL_CERT
: HPNA 2.0 certification protocol server code (ilc_cert.c).

DBG
: Expansion of debugging code (asserts, trace, log).

BCMDBG
: Some additional (Broadcom) debug code such as expanded dump output. BCMDBG is for code we wish to make available to our source code customers for debugging but that is off by default so that they don't ship it on (compiled in) by mistake.

DMA
: Support PCI DMA mode. Otherwise support PCI or MSI PIO mode.

MAXSE
: Support maxse payload encoding selection (rate) module.

Stat Counters

The driver allocates and maintains a common set of transmit and receive stat counters in the ilc_info_t structure. Total data memory storage requirements for these is around 200 bytes. Most of these are error counters and incremented by the software when the specific error is detected. Almost all are maintained by ilc.c. Some are maintained on and by the chip in hardware and read on demand before a chip reset and as part of responding to a "il dump" or port-specific stat counter request. All of the individual error counts that would potentially cause a transmit or receive frame to be dropped are summed on demand and reported as *txerror* and *rxerror* .

Each driver port provides a port-specific interface into these common core counters.

Chapter 3: Ethernet Driver Architecture

3.1 Introduction

This chapter provides an introduction to the internal design and implementation of the Broadcom BCM44xx Ethernet Networking device driver. The general structure and design methodology is presented along with a brief description of each *port* of the driver.

Since the device driver is evolving, the information presented here is subject to change without notice.

3.2 Background

The BCM44xx is an integrated controller for a 32 Mbps data-rate InsideLine HPNA 2.0, Ethernet IEEE802.3 networks, and a V.90 PCI/Mini PCI modem. It provides an integrated 10/100 Ethernet Media Access Controller (MAC) and

twisted-pair transceiver (PHY). BCM44xx provides a direct interface to the PCI bus, CardBus, or a Microprocessor Slave Interface for embedded systems. The chip supports 32-bit DMA and PIO (direct FIFO) accesses in PCI/CardBus modes and 16-bit PIO access, exclusively, in MSI mode, with full packet buffering. A 4096-byte 16/32-bit PCI memory space is decoded at BAR0. PCI IO space accesses are not supported by the chip. PCI function 0 is iLine, function 1 is Ethernet, and function 2 is V.90 modem. The digital chip currently interfaces with a separate, non-programmable, BCM41xx analog front end chip.

The term *DMA-mode* refers to the PCI master functionality of the chip, *PIO-mode* refers to direct programmed I/O use of the chip transmit and receive fifos independent of the bus (PCI or MSI), and *MSI-mode* refers specifically to the (PIO-only) Microprocessor Slave Interface functionality.

There are two DMA channels, one for transmit and one for receive. Each dma engine is configured via a set of four registers to process a contiguous vector (ring) of eight-byte descriptors. Descriptors are read-only and are never written by the chip. Each eight-byte descriptor consists of a four-byte control word and a four-byte 32-bit PCI address word. Each descriptor ring must be aligned on a 4096-byte memory boundary and, given the 4096 byte maximum contiguous dma segment size, each ring is limited to a maximum of 4096/8=512 entries. The chip supports arbitrary byte alignment and length of data buffers with the restriction that buffers may not span 4096 byte memory boundaries.

There are two 16-bit direct FIFO channels, one for transmit and one for receive. Each FIFO is accessed via a control and a data register.

There is a pair of 32-bit IntStatus/IntMask registers that are used in DMA mode and a pair of 16-bit MSIIntStatus/MSIIntMask registers that are used in PIO (PCI/CardBus or MSI -- not restricted to MSI) mode.

There is a Device Control register (*devcontrol*) at address 0x0 that functions as a master switch to enable the modem, transmit and receive FIFOs, and various test functions.

3.3 Chips Supported

The driver currently supports only the following chip:

- 4413 (vendor = 0x14e4, device = 0x4413), rev >=1 only

3.4 Ports List

The driver has been ported to the following operating systems and hardware platforms.

Name	OS	Hw Platform	DMA/PIO/MSI
NDIS	Win95	x86 PC	DMA/PIO
	Win98 and Win98SE	x86 PC	DMA/PIO
	NT4 (SP4)	x86 PC	DMA/PIO
	Win2000	x86 PC	DMA/PIO
	WinME	x86 PC	DMA/PIO
	WinXP	x86 PC	DMA/PIO (Beta 2 and above)
	CE v.2.12	BCM3310 MIPS	MSI
	CE v.2.12	x86 PC	DMA/PIO
	CE v.3.00	x86/MIPS	DMA/PIO
VxWorks	VxWorks 5.4	x86/MIPS	DMA/PIO
	VxWorks END	x86/MIPS	DMA/PIO
Linux	2.2 kernel	x86/MIPS	DMA/PIO/MSI
	2.4 kernel	x86/MIPS	DMA/PIO/MSI

3.5 Source Files

The driver is partitioned into OS-specific ("port-specific") and OS-independent ("common") source files. The naming convention is to use the **et**_ prefix for port-specific objects and **etc**_ prefix for common objects.

The Source package includes all of the common files and all of the port-specific files for a particular OS.

Common Files

etc.[ch]
: Principle "common" code. Shared amongst all port-specific ports of the driver. Most ports of the driver should not require modifying etc.[ch].

et_dbg.h
: Common header #includes the appropriate port-specific header file that defines tunables, printf/sprintf, bcopy/bcmp/bzero, register access macros, shared memory access macros, ASSERT, and trace/logging macros.

et_export.h
: Common header that defines the list of port-specific functions called by name from the common code. These functions must be provided by each port of the driver.

include/bcmenet.h
: Chip-specific manifest constants and data structures.

include/bcmendian.h

Defines a few platform-specific big/little-endian macros and #includes the native endian.h

include/typedefs.h

Driver-specific common typedefs (uint32, uint16, etc).

include/bcmdevs.h

Chip vendor and device identifiers.

NDIS Files

et_ndis.[ch]

main driver for NDIS 3.x/4.x/5.x

et_ndis30.[ch]

NDIS 3.0-specific shim functions

ce_ndis.[ch]

CE-specific shim functions

ndis41.h

Unedited local copy of the 4.1 version of NDIS.H used instead of installed 4.0 version when compiling for NT4.

et_ddk.[ch]

DDK/WDM-specific low-level functions

et_ddkCE.[ch]

CE/DDK-specific low-level functions

include/etioctl.h

NDIS custom OIDs used by the user-level **et** command

VxWorks Files

et_vxworks.c

VxWorks5.4 driver

et_vxworks.h

VxWorks5.4 specific header file.

include/etsockio.h

VxWorks5.4/Linux socket ioctls - used by the user-level **et** command

../VxWorks2_0_pc/

VxWorks5.4 Intel PC platform BSP directory

Linux Files

et_linux.[ch]

Linux driver

include/etsockio.h

socket ioctl definitions - used by the user-level **et** command

3.6 Data Structures

et_info_t (port-specific)

This is the principle, port-specific, per-device-instance private structure.

This per-port structure must begin with a common etc_info_t because callbacks from etc.c into the per-port routines cast their *etc_info_t** on the fly to the et_info_t. Typically, the et_info_t is tiny since it stores only OS- and hardware-platform-specific state.

Since this structure is port-specific, the common code (etc.c etc) can #include it.

Most port-specific routines take a et_info_t* as their first argument. The et_info_t pointer local variable is usually named *et*.

etc_info_t (etc.h)

This is the principle, common per-device-instance data structure. It is large and complex. Refer to the block comment and structure declaration in etc.h for more details.

Most common routines take a *etc_info_t** as their first argument. The etc_info_t pointer local variable is usually named *etc*.

bcmenetregs_t (bcmenet.h)

Chip registers plus symbolic macros for all the register bits.

bcmenetdd_t (bcmenet.h)

DMA Transmit/Receive Ring descriptor structure.

bcmenetrxhdr_t (bcmenet.h)

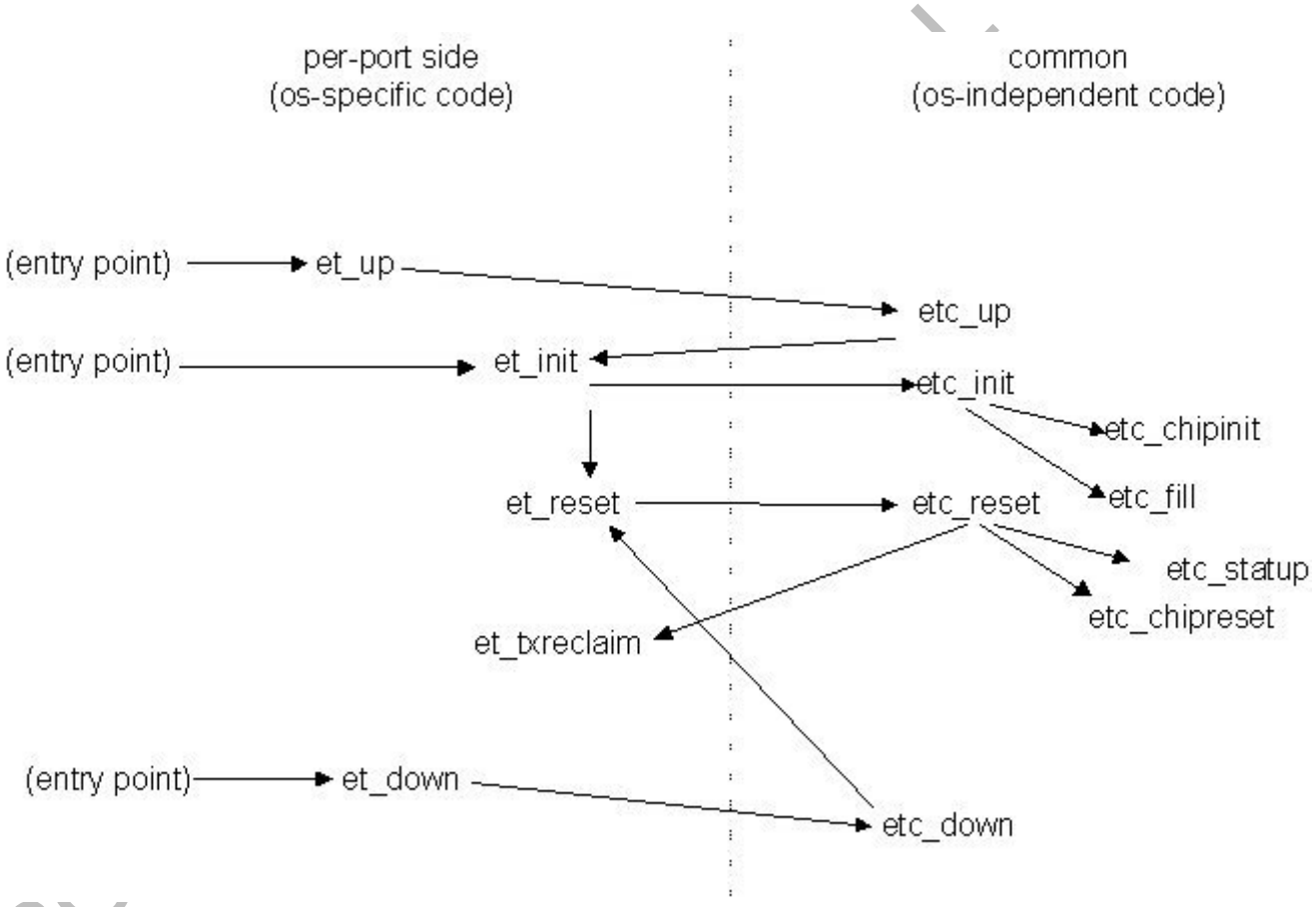
Received frames start with this 28-byte receive frame data header and consists of 16bits of frame length and 16 bits of EMAC rx descriptor, followed by 24 bytes of reserved data.

In PIO mode, these are the first bytes read by the driver from the receive FIFO. In DMA mode, this structure is placed (DMA written) by the chip at the start of the receive buffer.

3.7 Function Call Graphs

In each of the function call graphs below, the lefthand side contains the OS-specific functions and the righthand side contains the OS-independent functions. For clarity, some of the lower-level functions are not shown.

up/down/init/reset



The driver state is logically comprised as several layers -- rings of an onion -- in the following outer-to-inner order:

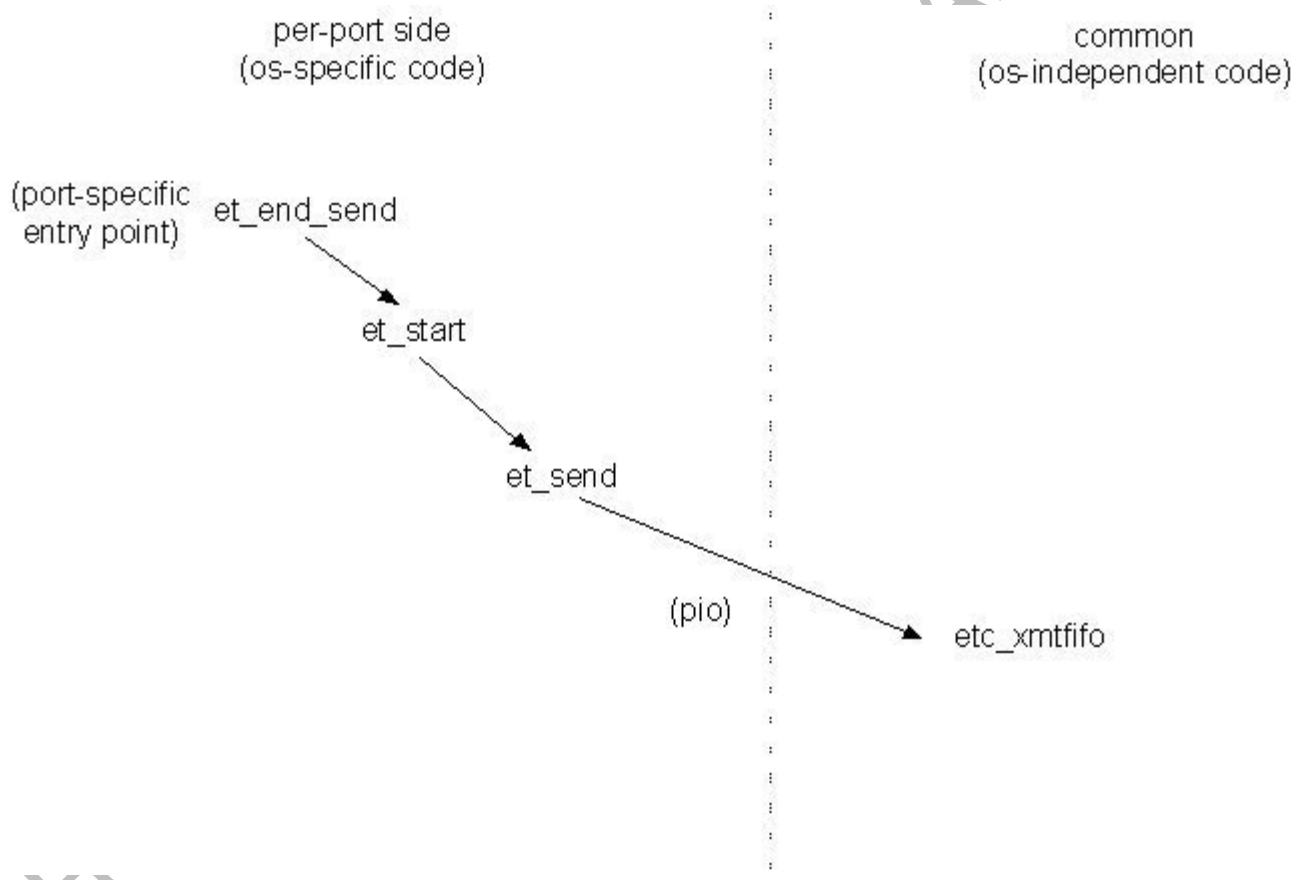
- driver load/unload
- device probe+alloc+attach/detach/free
- up/down - be persistently operational/non-operational
- init/reset - transient sw/hw re-initialization
- chipinit/chipreset - transient hw re-initialization

et_reset and *etc_reset* reset the logical sw/hw interface. *et_init* and *etc_init* reinitialize the logical sw/hw interface. *etc_chipreset* and *etc_chipinit* reset and performs most of the chip-specific re-initialization, respectively.

Most mode changes and error handling is accomplished via a reinitialization (et_init). This "big hammer" approach simplifies the code and greatly minimizes the number of possible state interactions.

Transmit

Because most of the drivers support both DMA and PIO modes as a runtime selection, some packet buffer queue is required in addition to the underlying chip transmit ring or FIFO. All driver entry points are port-specific. The following figure describes the VxWorks5.4 END driver case:

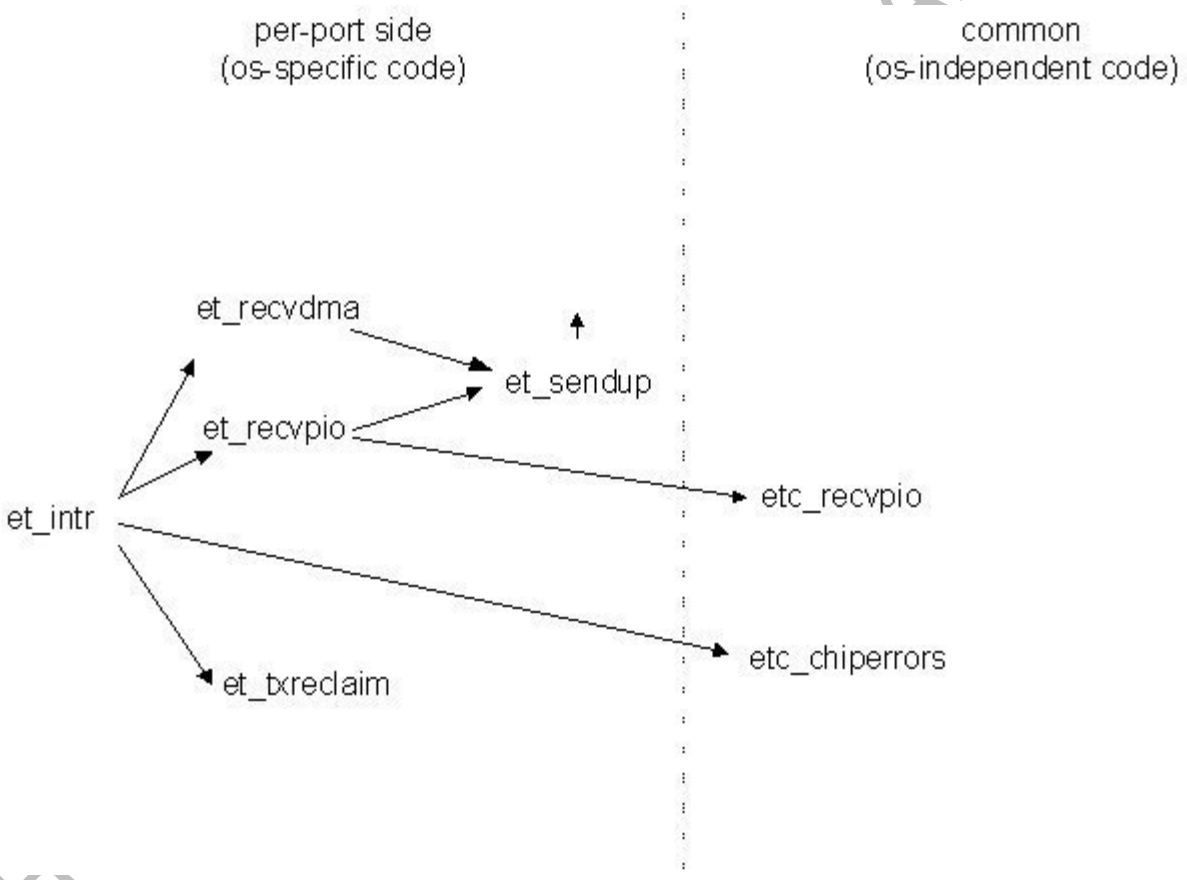


All drivers have a port specific entry point. In the case of VxWorks5.4 END driver it is *et_end_send*. It is *et_start* for Linux/vxWork5.4 BSD4.4 netif and *et_msendpackets* for NDIS drivers. NDIS and Linux have a private "sendnext" intermediate function. All ports always have an exported "send" hardware-transmit function.

The transmit packet gets enqueued at the at the port specific entry point and then *et_start* (in case of NDIS and Linux) *et_sendnext* is called to transmit the frame. *et_sendnext()/et_start()* determine if the internal resources required to transmit a frame are available. If they are, it calls *et_send()*. If they are not, it just returns. In PIO mode, the pointer pioactive indicates whether a transmit packet is already pending to the chip. In DMA mode, pioactive is always null. PIO-mode tx reclaim means just freeing the packet pointed to by pioactive. *etc_sendpio()/etc_xmtfifo()* handle the actual transmission in PIO mode. In DMA mode the hardware transmit is accomplished in *et_send()*.

The Linux driver, for example, processes sk_buffs that are single contiguous network buffers. The VxWorks driver processes chains of mbufs. None of these port-specific packet buffering structures can be accessed directly by the common code. Instead, the generic packet primitives (*il_pget, il_pfree*) are used.

Interrupt Handling



This figure shows the basic interrupt code path. The port-specific *et_intr* interrupt handler is a driver entry point called by the operating system. It identifies the particular interrupt event (tx, rx, error) and mode (dma, pio) and calls the appropriate port-specific or common function to process the event.

In PIO mode, transmit interrupts must be enabled. In DMA mode, transmit interrupts can be enabled or can be dynamically enabled or disabled, depending on port-specific buffering issues. For example, in DMA mode, the VxWorks and Linux drivers run with transmit interrupts disabled until the tx dma ring fills and the first packet is left on the software tx queue by *et_start()/et_sendnext()*. At this point transmit interrupts are enabled and remain enabled until the software tx queue is empty. The NDIS driver requires a more complex txreclaim scheme and so runs with transmit interrupts always enabled.

The et_txreclaim() routine reclaims any completed packets and starts the next one.

Either et_recvdma() or et_recvpio() is called to process received frames from the chip. In case of et_recv_dma() the received frame is processed and calls et_sendup() to send the packet buffer to the higher level protocols. In case of et_recv_pio(), etc_recv_pio() is called to process the packet prior to calling et_sendup().

Generic Packet Primitives

The per-port portion of the driver must provide the following set of packet primitives to allow common code to allocate, adjust, dup, and free native network packet buffers.

et_pget
: Allocate a native packet buffer of a given size. Returns a void* handle to the packet structure and, optionally, starting virtual and physical addresses of the data associated with the packet.

et_pfree
: Free a packet buffer specified by the void* handle.

Debug, Trace, and Log Primitives

The per-port portion of the driver must provide the following set of primitives to allow common code to support debugging and tracing operations. These functions are used when DBG or BCMDBG are defined.

et_assert
: Cause an exception when the condition is not true.

et_log
: Calculate the cycle and instruction count deltas from the previous et_log call. Call etc_log with the counts and the passed in parameters.

3.8 et/etc API

The driver is partitioned into port-specific and common source files. The interface between the port-specific and the common code in etc.c deserves some description because it is this interface that gets ported each time.

etc.c Exported Routines

Prototypes for the exported etc functions are declared at the bottom of etc.h. With rare exception, all etc functions take a etc_info_t* as their first argument.

Port-Specific Exported Routines

Common code cannot reference et_info_t since this data structure is port-specific. The common etc.c code makes some callbacks into the port-specific driver.

Those port-specific functions that are exported to the common code are declared in et_export.h:

```
        /* misc callbacks */
        extern void et_init(void *eth);
        extern void et_reset(void *eth);
        extern void et_txreclaim(void *eth, bool forceall);
        extern void et_rxreclaim(void *eth);
        extern void et_delay(uint us);
        extern void et_link_up(void *eth);
        extern void et_link_down(void *eth);
        extern int et_up(void *eth);
        extern int et_down(void *eth, int reset);
        extern void et_dump(void *eth, uchar *buf, uint len);

        /* canonical packet primitives */
        extern void *et_pget(void *eth, bool send, uint len, uchar
**va, uchar **pa);
        extern void et_pfree(void *eth, void *p, bool send);
```

3.9 Locking Model

Although the specifics vary, all ports share the following approach towards restricting thread preemption and concurrent execution:

- All data structures are enclosed within a single perimeter.
- This perimeter is enforced either (1) transparently by the operating system, (2) explicitly by the driver via a mutual-exclusion lock, or (3) a combination of the two.
- The perimeter lock is acquired and released by the port-specific entry points.
- The details of the perimeter lock are port-specific, therefore any state associated with the perimeter lock is maintained in the et_info_t data structure.
- The perimeter lock must be acquired before any common code is called (common code expects the perimeter lock to be held across the call).
- Common code never explicitly references the port-specific perimeter lock.
- The perimeter lock is *not* acquired recursively.

3.10 Port-specific Design Notes

The BSD and Linux drivers are more similar than different and almost identical at the functional level. Both are relatively small and simple compared to the NDIS driver or to the common etc.c code.

NDIS

For several reasons, the NDIS et driver is considerably larger and more complex than the other driver ports.

The NDIS et driver supports NDIS3 (Win95, WinCE), NDIS4 (NT4), and NDIS5 (Win98, Win98SE, Win2000, Millennium, WindowsXP). The NDIS and DDK/WDM variations are encapsulated either within #ifdef WDM within the principle et_ndis.c file or within separate small source files (et_ndis30.[ch], ce_ndis.[ch]). A small amount of necessary non-NDIS functionality is implemented within et_ddk.[ch] and et_ddkCE.c.

Two spinlocks are used - a general perimeter "lock" and a dpc-specific "dpclock."

There are two transmit NDIS packet queues, *txq* and *txdone*. All NDIS packets arriving at the MiniportSendPacket(s) entry point (*et_msendpackets*) are enqueued on txq. There is also a queue of local buffers *txctlq*, where the control packets (generated locally in the driver) are queued. The process of sending packets is accomplished in *et_sendnext*. *txctlq* is processed first, after which *txq* is handled. The txq queue has no flow control mechanism - it is of infinite size. After the packet has been successfully or unsuccessfully transmitted, it is placed on the txdone queue and the routine *et_sendcomplete()* called to NdisMSendComplete() all completed packets. Great pains are taken to meet the constraint that NdisMSendComplete() be called only from a Dpc/Timer routine with no private locks held.

There are two lists of free local buffers(*lbufs*), *txfree* and *rxfree*. At Miniport initialization, shared memory pages are allocated and split into one or more 2-Kbyte shared memory buffers to create each free list of local buffers. For transmit, the data comprising each original ndis packet chain is copied into a single, contiguous 2-Kbyte local tx buffer. For receive, the packet data is DMA'd directly into a single, contiguous 2Kbyte local rx buffer and the packet sent up to the higher level protocols.

The NDIS driver supports several features not provided in other ports of the driver:

Power Management (PM)
: NDIS-specific power management OIDs, programming the chip wakeup filter entries, entering and leaving D3 cold state.

Branding and Vendor-Specific Registry Issues

NDIS driver customers are more concerned with driver branding and providing vendor-specific options such as changing the default values for link level procotols and modes based on registry values, and querying arbitrary OEM data contained within the hardware serial PROM.

WinCE

WinCE implements a variant of the NDIS3.0 interface. The Windows CE variations to the NDIS driver are contained within the source files ce_ndis.[ch], ce_ddkCE.c and the inline conditional #ifdef UNDER_CE'.

VxWorks

The VxWorks driver, *et_vxworks.c*, currently supports VxWorks 5.4 END and BSD4.4 netif driver models.

Transmit mbuf chains are enqueued on the if_snd queue and drivers *et_start()* routine called which could be renamed *et_sendnext()* since it is serving in this capacity. There are no private buffers allocated nor maintained in the driver. DMA reads and writes are done directly from/into the transmit and receive mbufs, respectively.

The source file *etsockio.h* defines several et driver ioctl command values used by the Linux, and VxWorks drivers.

The perimeter thread exclusion is implemented via semphore-based splimp()/splx() for BSD4.4netif and using a private mutex lock for END driver model. In both VxWorks driver models the hardware interrupt entry point merely turns off the bits causing the interrupt and creates a task that does the actual interrupt handling at task level.

Linux

et_linux.c currently supports the 2.2 and 2.4 linux kernels and MODULE loading.

No private buffer pools are needed. *et_start()* is the transmit entry point and enqueues the skb onto the private txq. *et_sendnext()* determines if the tx ring has available slots in DMA mode, or pioactive is NULL in PIO mode, before dequeueing the skb from the private txq. *et_sendnext()* also monitors the software txq to dynamically enable/disable tx interrupts when necessary and to set and clear the tbusy flag which flow-controls the upper protocol layers. Since skbs do not chain, *et_send()* is almost trivial.

Received frames are DMA'd directly into allocated sk_buffs or are PIO copied directly from the Receive FIFO register into an sk_buff (by calling *etc_recvpio()*) before *et_sendup()* is called to deliver the packet to the protocol layers.

A single spinlock (spin_lock_irqsave()/spin_unlock_irqrestore()) is used to singlethread the majority of the driver.

The Linux UDP implementation implements driver transmit-side flow-control. At most, **sockbuf** number of frames are allowed to be pending for transmit at any one time. This requires that the linux et driver run with transmit interrupts enabled and not be too lazy in reclaiming completed transmit packets.

3.11 Other Miscellaneous Functionality

Probe/Attach/Allocation

The mechanics of driver registration, loading, and probing, tend to vary from port to port. Linux drivers support a loadable module interface.

After the device has been detected, the sequence of operations tends to be similar:

- Allocate et_info_t memory and bind this to the ifnet/device structure.
- Map chip registers.
- Confirm that the chip is one of ours.
- Read the chip revision.
- Reset the chip.
- Initialize spinlock or transmit queues.
- Register our interrupt handler.
- If in DMA mode, allocate tx and rx descriptor rings.
- Call etc_attach which checks that the BCM44xx chip is accessible and performs some common attach-time initialization.
- Allocate and init the grotty stats data structure.
- Optionally, set/override the default local Ethernet address.
- Enable our entry points.

If any of these operations fail or we support unloading the driver, the et_free routine gracefully deallocates all of this.

Conditional Compile Flags

The following driver-specific, OS-independent compile-time flags are used:

DBG
: Expansion of debugging code (asserts, trace, log).

BCMDBG
: Some additional (Broadcom) debug code such as expanded dump output. BCMDBG is for code we wish to make available to our source code customers for debugging but that is off by default so that they don't ship it on (compiled in) by mistake.

DMA

Support PCI DMA mode. Otherwise support PCI or MSI PIO mode.

Stat Counters

The driver allocates and maintains a common set of transmit and receive stat counters in the etc_info_t structure. Total data memory storage requirements for these is around 200 bytes. Most of these are error counters and incremented by the software when the specific error is detected. Almost all are maintained by etc.c. Some are maintained on and by the chip in hardware and read on demand before a chip reset and as part of responding to an **et dump** or port-specific stat counter request. All of the individual error counts that would potentially cause a transmit or receive frame to be dropped are summed on demand and reported as *txerror* and *rxerror*.

Each driver port provides a port-specific interface into these common core counters.

Performance Measurement Tool ET_LOG

This exactly identical to IL_LOG. Please see section 5.3 IL_LOG Performance Measurement Tool.

3.12 et dump Example

The et command when run with dump option provides a lot of useful information. The following is an example of the output is described.

The first section is the port specific portion and consists of fields from et_info structure.

```
et0: Jun 13 2001 15:32:53 version 2001.4.25.0 (BROADCOM INTERNAL)
et 0x7f6db6c bpfext 0 if_flags 0x8063 ifq_len 0
```

The next section consists of fields from etc_info_t structure. For example, the link is up (up 1) and one multicast address is set (nmulticast 1 and multicast addr list: 1:0:5e:0:0:1).

```
etc 0x7f6db6c regs 0xfebbf000 msglevel 1 speed/duplex 100full
up 1 promisc 0 loopbk 0 forcespeed -1 advertise 0x0 needautoneg 0
piomode 0 pioactive 0x0
nmulticast 1 allmulti 0
vendor 0x14e4 device 0x4412 chiprev 1
subvendor 0x14e4 subid 0x105 phyunit 1
perm_etheraddr 0:90:4c:c:3:1 cur_etheraddr 0:90:4c:c:3:1
multicast addr list: 1:0:5e:0:0:1
```

```
      txd 0x7f69000 txdpa 0x7f69000 txp 0x7f6dc94 txin 0 txout 0
      rxd 0x7f6a000 rxdpa 0x7f6a000 rxp 0x7f6e0a4 rxin 0 rxout 32
      intmask 0x1fc00 dogtimerset 1 linkstate 1
```

The next section constitutes the various software stat counters maintained by the driver (aggregate since boot).

```
      txframe 1 txbyte 42 txerror 0 rxframe 0 rxbyte 0 rxerror 0
      tx_broadcast_pkts 1 tx_multicast_pkts 0 tx_jabber_pkts 0
tx_oversize_pkts 0
      tx_fragment_pkts 0 tx_underruns 0
      tx_total_cols 0 tx_single_cols 0 tx_multiple_cols 0
tx_excessive_cols 0
      tx_late_cols 0 tx_defered 0 tx_carrier_lost 1 tx_pause_pkts 0
      txnobuf 0 reset 5
      rx_broadcast_pkts 0 rx_multicast_pkts 0 rx_jabber_pkts 0
rx_oversize_pkts 0
      rx_fragment_pkts 0 rx_missed_pkts 0 rx_crc_align_errs 0
rx_undersize 0
      rx_crc_errs 0 rx_align_errs 0 rx_symbol_errs 0 rx_pause_pkts 0

      rx_nonpause_pkts 0 rxnobuf 0 rxdmauflo 0 rxoflo 0 rxbadlen 0
```

The last section is a dump of the hardware registers.

```
      devcontrol 0x60 devstatus 0x44130201 wakeuplength 0x0
      intstatus 0x0 intmask 0x1fc00 gptimer 0x0
      enetaddrlower 0x4c0c0301 enetaddrupper 0x90
      emaccontrol 0x9 emacflowcontrol 0x0
      gpiooutput 0x0 gpioouten 0x0 gpioinput 0x3ffff
      funcevnt 0x0 funcevntmask 0x0 funcstate 0x2
      intrecvlazy 0x1000000
      xmtcontrol 0x1 xmtaddr 0x7f69000 xmtptr 0x0 xmtstatus 0x2000
      rcvcontrol 0x3d rcvaddr 0x7f6a000 rcvptr 0x100 rcvstatus 0x1000
      rxcontrol 0x60 rxmaxlength 0x60e txmaxlength 0x60e
      mdiocontrol 0x9f camcontrol 0x10001 enetcontrol 0x1
      txcontrol 0x1 txwatermark 0x38 mibcontrol 0x1
      tx_good_octets 0x0 tx_good_pkts 0x99 tx_octets 0x0 tx_pkts 0x99
      tx_broadcast_pkts 0x0 tx_multicast_pkts 0x0
      tx_jabber_pkts 0x0 tx_oversize_pkts 0x0 tx_fragment_pkts 0x0
      tx_underruns 0x0 tx_total_cols 0x0 tx_single_cols 0x0
      tx_multiple_cols 0x0 tx_excessive_cols 0x0 tx_late_cols 0x0
      tx_defered 0x0 tx_carrier_lost 0x0 tx_pause_pkts 0x0
      rx_good_octets 0x0 rx_good_pkts 0x37d8 rx_octets 0x0 rx_pkts
0x37d8
      rx_broadcast_pkts 0x0 rx_multicast_pkts 0x0
      rx_jabber_pkts 0x0 rx_oversize_pkts 0x0 rx_fragment_pkts 0x0
      rx_missed_pkts 0x0 rx_crc_align_errs 0x0 rx_undersize 0x0
      rx_crc_errs 0x0 rx_align_errs 0x0 rx_symbol_errs 0x0
      rx_pause_pkts 0x0 rx_nonpause_pkts 0x0

      phy0 0x3000 phy1 0x782d phy2 0x40 phy3 0x6000
      phy4 0x1e1 phy5 0x41e1 phy24 0x3f phy25 0xf53f
```

3.13 Porting the Driver

Broadcom supplies sample drivers for VxWorks5.4 (END and BSD4.4 netif), Linux, and Windows NDIS (sources only for CE) driver models. The drivers consist of a OS-independent portion etc.c and a OS-specific portion (et_linux.c, et_ndis.c and et_vxworks.c). You do not need to modify etc.c. If you are porting for one of the above operating systems you must modify only the OS-specific portion.

In general you must start with closest matching sample code and do the following:

1. Add support for OS in typedefs.h.
2. Modify macros in et_.h.
3. #include et_.h in et_dbg.h.
4. Specify et_info_t for your os-port.
5. Create all functions in et_export.h.
6. Create equivalent functions for those in et_.h.

Chapter 4: HPNA 2.0 Control Protocols

4.1 InsideLine Control Protocols

There are several link-layer control protocols which share an *ILCP* Ethernet frame type field value of 0x886c. Following this 16-bit ILCP ether_type is a 16-bit ILCP subtype field with the first eight-bit *subtype* field indicating the particular ILCP sub-prototocol and the next eight-bit *length* field indicating the sub-protocol header length. The header file *proto/ilcp.h* defines ILCP symbolic structures and manifest constants.

Refer to the HPNA 2.0 specification for more details on link-layer control protocols.

Rate Negotiation

Each receiver actively monitors the signal-quality of received frames and dynamically transmits ILCP Rate control frames back to the source station advertising the appropriate transmit constellation (bits-per-baud) to use. Generally the "bits-per-baud" value is referred to instead of the encoded frame constellation value which includes attributes in addition to the bpb. Transmitting stations are "dumb" and simply use the most recently bpb value received from

the remote station. Currently supported bpb values are in the range 2-8. Rate control frames have a subtype value of 1.

Link Integrity

Each station broadcasts a small ILCP link integrity frame approximately once each second. There is also a mechanism defined by the spec which suppresses excess link integrity control frames when there are more than two stations active on a network. Link Integrity control frames have a subtype value of 2.

CSA

Each station is required to broadcast a Capability and Status Announcement control frame approximately every 60 seconds and on certain events. CSA control frames contain a few bytes of relatively static information about the station version and capabilities plus three 32bit {cur_tx, old_tx, cur_rx} priority and mode flags. At present, CSA is used by our driver to provide two functions: (1) priority mapping and (2) network operating mode override.

Eight levels of physical media priority (1-8) are provided in the HPNA 2.0 spec and implemented in iLine10. Due to the contention slottime size, *as a performance optimization*, logical priority values are dynamically mapped into the highest N physical priority values where N is the number of logical priorities used (transmitted) by any station within the past 120 seconds. Each station sends a CSA control frame whenever a new priority is transmitted, or every 60 seconds. Each station maps the logical-to-physical priority on transmit based on the union of all received priority sets in the past 120 seconds. It's a little more complicated than this but not much.

CSA also provides a method to force all stations on the network to Native (iLine10), Mixed-mode (compat, gapped), or Tut (HPNA 1.0). This overrides the dynamic algorithms used on each station and is analgous to using a baseball bat.

Refer to the ilcp.h header file for CSA definitions. CSA control frames have a subtype value of 3.

LARQ

LARQ (Limited Automatic Repeat Request) is a protocol that reduces a network's effective error rate by supporting the detection and retransmission of errored or dropped frames. Its goal is to significantly enhance the usability of networks that may, at least occasionally, have frame error rates (FER) of 1 in 10^-2 or worse. The protocol is based on a Negative Acknowledgement (NACK) mechanism. LARQ functions as an adaptation layer between the link layer (Ethernet) and the network layer (IP). An OS-independent implementation is provided in the source

files plarq.h and plarq.c. LARQ control frames and LARQ-encapsulated data frames have a subtype value of 4.

Chapter 5: Porting the Code

5.1 Porting Process

Porting the HPNA driver to a new operating system or hardware platform is typically a two-step operation:

1. The BASIC port supports basic frame transmittion and reception and is verified using the port validation test scripts.
2. The FULL port add support for InsideLine control protocols (ILCP) (RATE, CSA, and LARQ) and is verified using the same port validation test scripts with these new protocols enabled.

For details about the InsideLine control protocols, refer to the HPNA 2.0 specifications. You must become an HPNA member to obtain the specifications. Please see the HomePNA Web site at *http://www.homepna.org/* for details.

Design Considerations (Pre-Port)

- System design should be based on the cpu and memory requirements of the FULL driver, NOT on the cpu and memory requirements of the BASIC driver.
- Approximately 128 Kbytes is used by the BASIC InsideLine device driver. An additional 128 Kbytes to 512 Kbytes is required for the InsideLine Control Protocols supported in the FULL version.

Note: Insufficient buffer memory can lead to decreased performance in noisy (packet dropping) environments.

For a fully HPNA 2.0 compliant system, the current driver requires approximately 10-15 MIPS (for embedded source release 2.33 and beyond) while transmitting minimum sized frames (70 bytes) at the maximum media rate of 8000 frames/s . This can vary ±20% based on the compiler/processor and memory latency. Also, these numbers reflect an MSI interface (programmed IO) which is common in embedded designs. CPU utilization in PCI/DMA mode is substantially better (lower) than PIO mode.

Porting Guidelines

The OS-independent portion of the il device driver is provided as a *black box* and should not require modification by the customer. The focus of the porting process

should be to implement and debug the driver OS-specific functionality. This approach helps insure interoperability with other HPNA devices.

It is important to understand that the OS-independent or "common" code affects HPNA 2.0 compliance and, therefore, should not be modified.

The source code modules for BCM42xx Embedded drivers (pSOS, VxWorks, Linux, and WinCE) that are part of the "black box" are listed below:

Black Box Drivers

Basic Driver Black Box Code	Full Driver Black Box Code
crc.c	crc.c
ilc.c	ilc.c
ilc_cert.c	ilc_cert.c
N/A	pe_select_maxse.c
N/A	plarq.c

Note: If you feel you must modify the "common" code, you MUST consult with Broadcom about the appropriateness of the change and its implications.

Driver Port - Phase I

Each driver port must implement the following port-specific API functions. A C language prototype of each function is provided in *il_export.h*. The port-specific functions are exported to common code and do not require that an il_info_t* argument is declared in ilc.h.

Broadcom driver source releases contain fully functional port-specific files for pSOS, VxWorks, WinCE and Linux. In most cases, these can be used with little or no modification. For other operating systems, one of these ports can be used as a starting point.

API Functions Required for Port

Function Name	Description
il_reset()	Reset the driver and chip
il_init()	Reset and reinitialize the driver and chip
il_sendctl()	Send an ILCP frame
il_txreclaim()	Reclaim all completed transmit descriptors
il_rxreclaim()	Reclaim all active receive descriptors

Function Name	Description
il_sendup()	Pass received frames up to higher-layer protocols
il_ms()	Return the current time in milliseconds
il_delay()	Spin delay some number of microseconds
il_link_up(), il_link_down()	Indicate link up/down
il_up()	Reinitialize and mark the interface operational
il_down()	Reset and mark the interface non-operational
il_dump()	Formatted printf of the driver and chip state
il_malloc(), il_mfree()	Allocate/free memory
il_pget()()	Allocate a native packet buffer of a given size
il_pfree()	Free a native packet buffer
il_ppush()	Add N bytes to the front of a native packet buffer
il_ppull()	Remove N bytes from the front of a native packet buffer
il_plen()	Return the total number of bytes in the native packet buffer
il_psetlen()	Set the total number of bytes in the native packet buffer
il_pdup()	Duplicate and return a copy of a native packet buffer
il_larq_add_timer()	Schedule a non-periodic LARQ timer
il_larq_del_timer()	Deschedule a non-periodic LARQ timer

Driver Port - Phase II

After you complete the Phase I port and believe you have a functioning BASIC InsideLine driver, Broadcom provides a set of Port Validation Scripts (PVS) that allow you to exercise some aspects of the driver.

You must first run a BASIC validation test on your system to verify your system's HPNA throughput and packet error rate (PER) performance. A PVS users guide and Test Script Package is available separately. The minimum hardware required for the PVS is:

- Two NT 4.0 (Service Pack 5 or higher) or Win2k systems
- PCI HPNA 2.0 NICs and recent Broadcom device drivers installed in each machine
- Customer Device Under Test.

After successfully transferring tens of megabytes of data in both directions, please provide the output from "il dump" to Broadcom for review of throughput and packet error rates.

The Port Validation Scripts (PVS) test the following:

PER test (Packet Error Rate)

This test reports the total number of frames lost. A Passing threshold is less than 1%. The frame rate is 8 frames each 16 milliseconds which is much slower than the maximum media rate. The two frames sizes used are (256 bytes at 2 bpb) and (1024 bytes at 6 bpb).

Throughput test

This test measures and reports both transmit and receive data throughput at various combinations of frame sizes and transmit payload encodings. There is no Pass/Fail.

Enhanced Throughput test (RAMP)

This newer test sends a programmable number of iLine control packets back-to-back to the DUT from a controller node. This test measures device receive performance only. It "ramps" through different packet lengths at different payload encodings. It then checks how many packets are successfully received by the DUT under variable traffic conditions. At the end of the test, it shows the data point for each packet length for a specific PE to plot the performance curve. The controller node can be either Windows NT4.0 (SP5 or better) or Win2K.

After the results of the PVS are provided, Broadcom can analyze the dump results and provide recommendations as necessary on improving performance.

Driver Port - Phase III (Going from BASIC to FULL Port)

Converting from BASIC to FULL requires replacing the BASIC common code version with the FULL common code version that includes ILCP functionality. Once resource and CPU power requirements are met and you have made very little or, ideally, NO MODIFICATIONS to the common code, you should be finished porting at this phase and ready for the Logo submission to HPNA TAB (see below).

Per-Port Advice

- Do not remove ASSERTs -- they provide validation of a proper driver port and help improve stability and reliability.
- Do not remove reliability checks, such as register accessibility or SPROM content validation.
- Do not remove the perimeter locking mechanism (spinlock, splhi, etc). This can introduce serious runtime errors.
- Do not gratuitously remove code. At the very minimum it will make your job harder when upgrading to future releases.

Porting for Bridge Applications

To make the Broadcom HPNA driver work with a target bridge device you must:

- Enable the IL_BRIDGE #define (-DIL_BRIDGE) .
- Implement *bridge_peek()* .

bridge_peek() function

```
extern int bridge_peek(void *h, uint8 *ea);
```

bridge_peek() is called by the driver to determine whether the bridge is acting as an ILCP proxy for a particular remote station. bridge_peek() returns nonzero (TRUE) if the Ethernet address passed as an argument is recognized as an address of a station on the local HPNA network. If the Ethernet address is not recognized, zero is returned. This return value does not depend upon whether or not the bridge will actually forward the frame.

The *h* argument is an opaque interface-specific handle. If the system has more than one HPNA interface, this handle is used to determine which HPNA interface was making the bridge_peek() call.

5.2 Porting To-Do list

Don't Panic.
-- Douglas Adams, "Hitchhikers guide to the galaxy"

Starting with the closest per-port driver above, you must resolve the following issues:

1. Add support for your OS to typedefs.h.
2. Create a port-specific header file (name convention is il_*foo*.h) for the following. Include the appropriate native header files and define the following as cpp macros, declarations, or static inline functions.
    - required common code tunables (NTXD, NRXD, etc.)
    - (optional) private driver-specific tunables
    - printf() and sprintf() declarations
    - R_REG(), W_REG(), AND_REG() and OR_REG() register access definitions
    - bcopy(), bcmp(), and bzero() declarations
    - R_SM(), W_SM(), and BZERO_SM() shared (dma-able) memory access definitions
3. Add an #include for your port-specific header file to il_dbg.h .
4. Create your port-specific il_info_t declaration. This must start with an ilc_info_t. You might write a il_dump() now which calls ilc_dump() for all the common state and verifies register accessibility.
5. Write your port-specific driver load/unload/probe/attach gunk code using the BSD driver il_attach() routine or the Linux driver il_probe1() routine as reference. If your platform is MSI (not PCI) or does not include an

SPROM, then you must deal with setting the ilc regs, cur_etheraddr, perm_etheraddr, aferev, subid, and subvendor fields.

6. Write il_up, il_down, il_init, and il_reset next because they are pretty simple and the driver/chip can at least init now.
7. Arrange a periodic timer to call il_watchdog() every 1 second. Put a debugging printf() in il_watchdog() to confirm that it is being called when the driver is "up," and not when the driver is "down."
8. Next you can write the transmit code path, starting with the transmit driver entry point, including il_sendnext(), il_send(), il_sendctl(), and il_txreclaim(). If you are doing DMA-only or PIO-only, you can cut corners slightly here. PIO functionality is simpler than DMA.
9. Hardware platforms interfacing to the chip using PIO/MSI bus benefit by having highly optimized direct FIFO access routines for transmit and receive. You can write port-specific optimized replacements for ilc_recvpio() and ilc_xmtfifo() now.
10. Next you can write the receive code path routines, beginning with your port-specific interrupt handler (il_intr in the case of the BSD and Linux drivers), then il_recvdma() and/or il_recvpio(). Most of the receive work is done by the common ilc_recv() routine, but il_recvpio/dma has to deal with port-specific packet buffers.
11. Implement all of the il_pxxx() packet primitive routines {il_pget, il_pfree, il_ppush, il_pull, il_plen, il_psetlen, il_pdup}. ASSERTs are your friends here.
12. Support LARQ by implementing il_larq_add_timer() and il_larq_del_timer() functions atop your native timer facility. Use a debugger or printf to verify that these timers are working correctly.
13. There are several miscellaneous routines required by ilc.c, like il_ms(), which returns the current time in milliseconds, il_delay() which takes the number of microseconds to spindelay, and il_malloc()/il_free(), which allocate and free a chunk of vanilla non-dma-able memory.
14. Implement support for adding and removing discrete multicast addresses or "all multicast" addresses.
15. Modify include/epivers.h.in to provide the appropriate information about your product. Valid values for HPNA_DEV_TYPE are listed in ilcp.h.

    HPNA_VENDOR
    HPNA_DRV_BUILD_DATE
    HPNA_HW_MFG_DATE
    HPNA_DEV_TYPE

16. You might need an ioctl handler and user-level command that issues the ioctls. This is very useful in debugging.

Testing Your Port

1. load/unload/probe/attach. Be aware that the ilc_attach routine starts by testing that the BCM42xx chip is accessible and it will throw an assert very early if it fails to read or write and read-back some of the chip registers. Look at the routine validate_chip_access() at the end of ilc.c.
2. up/down - try il_dump() to ensure that software and chip state have been properly initialized. Verify link integrity functionality.
3. transmit/receive path - ping, "ttcp" or comparable test app. A memory-to-memory TCP transfer between two stations over a point-to-point link should show a throughput in the 1.2-1.3 Mbytes/s range using a negotiated rate of 7 or 8 bpb.
   Then send "hello world" email to management :-)
4. Two stations exchanging InsideLine frames over a short point-to-point twisted pair wire should quickly negotiate to using transmit bits-per-baud of 7 or 8 (2Mbaud payload encoding values 6 or 7) in both directions. Use "il scbdump" to print the contents of the driver SCB table. The values under "txd" and "rxd" are payload encodings. If built with -DDBG, the IL_TRACE(), IL_ERROR(), IL_LOG(), and IL_PRHDRS() debugging macros are enabled with various values of "il_msg_level."
5. Verify periodic CSA control frame transmit and whenever a new transmit priority is used. ilc_dump() prints the local CSA state fields.
6. Stress test while resetting and introducing error frames and check for memory leaks.
7. mode switching - add an HPNA 1.0 station to the net and monitor the driver "macmode" and "pcom" fields. Try an "il hpnamode [0-3]" to force the local station into each of the operating modes while monitoring the format of link integrity and data frames. Try a "il csahpnamode [0-3]" to force all stations on the net to a particular operating mode and back.
8. Run the Port Validation Scripts (PVS).

5.3 IL_LOG Performance Measurement Tool

Our drivers come with some tools to help measure and improve their performance. CPU cycle and instruction counts can be accurately measured on some platforms, and performance improvements can be tracked as changes are made to the code.

Measuring performance starts with our logging functions. In our iLine driver, there is a macro called `IL_LOG()` that captures the current instruction count and cycle count.

Initialization

The function `il_loginit()` must be called once to perform any platform-specific initialization. All logging functions are conditionally defined with the `BCMDBG` pre-processor symbol. Unless this symbol is defined during compilation, all of the logging functions mentioned here will be NO-OPs. Furthermore, the logging

functions are enabled at runtime by setting bit 11 of the global variable `il_msg_level`, defined and initialized in `ilc.c`. Be sure to set `il_msg_level` before calling any logging function.

```
        il_msg_level |= 2048;
        il_loginit();
```

Capturing Performance Information

The basic method is to capture counters before and after a given operation and subtract first measure from the second to come up with the cost of the operation. For example, if you are interested in measuring the cost of the routine `il_sendup`, the code might look something like this,

```
        IL_LOG("il_sendup", 0);         /* capture counters before you
call il_sendup... */
        IL_LOG("il_log", 0);            /* used to find logging overhead.
*/
        il_sendup((void*)ilc, tmp, pri);
        IL_LOG("il_sendup ret\n", 0);  /* ...and capture counters after
you call il_sendup */
```

When measuring performance of a routine, you typically bracket the routine with two calls to IL_LOG, the first of which is tagged with the name of the routine or code segment, and the second of which repeats the name with "ret" appended to the end. This syntax is not just a good idea, it is essential to the correct operation of the analysis tools that are run later. In this example, the tags for "il_sendup" follow that format.

The two arguments from `IL_LOG` are stored along with the performance counters captured at that point. Each call to il_log results in an entry in a circular buffer which can store 256 entries before overwriting. When the circular buffer is dumped, the arguments passed to il_log are passed to printf along with the value of the performance counters.

As with the other logging functions, `IL_LOG` only works if the driver was compiled with the `BCMDBG` preprocessor symboled defined, and if `il_msg_level` has bit 11 set at the time `IL_LOG` is called.

Collecting Performance Information

Once you have run your sample code and presumably collected some log entries in the log buffer, you will want to dump the log buffer to see the results. The contents of the circular log buffer can be retrieved with a call to `ilc_dumplog`. This routine takes a single argument which should be a pointer to a big buffer (at least 4K) into which the contents of the log buffer are be formatted.

The formatted string produced by `ilc_dumplog()` looks like this:

```
67268   2734    il_recvdma
248     58      il_log tot/dma/maxse
518     83      framelen 71
910     296     ilc_rateupd
646     188     ilc_rateupd ret
2770    936     il_recvdma ret

68064   2744    il_recvdma
248     58      il_log tot/dma/maxse
522     83      framelen 71
894     296     ilc_rateupd
524     199     pe_select_update_stats
644     389     pe_select_update_stats ret
446     126     ilc_rateupd ret
3048    936     il_recvdma ret

67008   2734    il_recvdma
248     58      il_log tot/dma/maxse
518     83      framelen 71
888     296     ilc_rateupd
660     188     ilc_rateupd ret
2800    936     il_recvdma ret
```

After calling `ilc_dumplog()`, the returned ASCII string can be printed to a file or to the serial console.

Each entry in this log consists of the number of cycles since the previous entry, the number of instructions since the previous entry, and the tag as specified by the arguments passed to the IL_LOG macro. Both counters may not be available on every platform, but at least one of these is usually available on most platforms.

Analyzing the Data

The simplest way to analyze the performance data you have collected is to look at the raw log. Each entry contains the number of instructions and cycles since the previous entry. Let's look at the data for just one frame from the log above.

```
67008   2734    il_recvdma
248     58      il_log tot/dma/maxse
518     83      framelen 71
888     296     ilc_rateupd
660     188     ilc_rateupd ret
2800    936     il_recvdma ret
```

From this example, you can see that 888 cycles and 296 instructions elapsed in the code fragement labelled "`ilc_rateupd`". That means the between the

previous entry for "framelen" and the one for "ilc_rateupd", 296 instructions were executed. It does not mean that the "ilc_rateupd" routine itself took 296 instructions; that sort of data must be derived later using post-processing tools.

Finding out how many instructions were executed for all of "ilc_rateupd" is a little tricky. To do that you must add up all the instructions counts for all the log entries between "ilc_rateupd" and "ilc_rateupd ret", not including the first one (because that represents the count from the previous tag). In this example there are no tags between "ilc_rateupd" and "ilc_rateupd ret", so our count is simply the count for "ilc_rateupd ret", 188 instructions.

To perform the same operation for "il_recvdma", you must add together all the instruction counts between "il_recvdma" and "il_recvdma ret", again, excluding the first entry. That would be 58+83+296+188+936=1561 instructions. Note that you do not count the very first entry for "il_recvdma" because that pertains to the number of instructions since the *previous* entry.

One thing that our calculation has not taken into account is the overhead for the logging function itself. The cost of the log function can be calculated simply by putting two log statements back-to-back in the source code, e.g.

```
IL_LOG("il_recvdma", 0);
IL_LOG("il_log %s", "tot/dma/maxse");
```

That is what was done to create the log sample above. The "il_log" entry in the sample above contains just the count of cycles and instructions between sequential calls to the log function, which is actually the same as the cost of calling the log function itself.

Now you can go back and recalculate the performance of our code by taking into account the cost of the log function itself. To do that, reproduce the log entries after subtracting the the cost of the "il_log" entry.

```
64407   2676    il_recvdma
0       0       il_log tot/dma/maxse
298     25      framelen 71
745     238     ilc_rateupd
443     130     ilc_rateupd ret
2679    878     il_recvdma ret
```

By repeating our calculation for "il_recvdma" (0+25+238+130+878=1271) you see that the previous calculation included total overhead of 22% (1561 vs. 1271). Good thing the cost of logging was taken into account.

A note about interpreting the cycle count results - The cycle counts can be wildly variable, sometimes differing by as much as 100% from run to run. On the other

hand, instruction counts are highly reliable. For the same input under the same conditions, the instruction count should not vary at all from run to run. If they do, it is likely that some interrupt service routine or other asynchronous event happened during your performance measurement.

Cycle counts can be influenced by a wide set of subtle factors, like the speed of memory, cache misses, other applications in memory, etc. Over very large runs (hundreds, if not thousands) cycle counts can still provide a good relative of performance. In fact cycle counts are sometimes the only way to reduce the number of cache misses if direct observation of the cache miss count is not possible. But the high variability of cycle counts make them useless for fine-grained tuning.

Advanced Analysis

Analyzing raw log entries is a tedious and error-prone process. That is why we have developed a `PERL` script to help. The script automates a tremendous amount of the drudgery associated with processing raw performance logs but it requires some specific conventions in order to work correctly.

To invoke the log processing script, you would issue a command like,

```
perl perf.pl iline.log
```

`iline.log` is the name of a file that contains the textual data dumped from "`ilc_dumplog`". The script will produce four output files with same basename as the input file, of which only two are actually useful. The other two contain what is essentially debugging information for the log processing script.

`iline.dtl` - Details of per packet performance.

`iline.tbl` - Packet table

`iline.tag` - Tag table

`iline.ptl` - Per-packet instruction counts

`iline.dtl` - Details

```
il_intr          892      892    100%
_il_recvdma              863      892    96%
__*framelen      27       892    3%
```

The **`Details`** file contains a call graph profile of the logged functions. Each line contains the tag name, the total instructions for that tag and its descendents, the total instructions for the packet, and the percentage of the total represented by this tag and its descendents.

One tag is considered to be a descendent of another if it appears within a "parent_tag/parent_tag ret" pair. Tags that do not have any corresponding "tag ret" entry are marked with a '*'. Each level of descendent is preceeded with an additional '_'.

There is one tag that never shows up in the detail listing, and that is the `il_log` tag. If the `il_log` tag is present in the log for a given packet, the instruction count and cycle count for that tag are subtracted from every tag entry to account for logging overhead.

`iline.tbl` - Packet Table

len	Imin	Imax	Imean	Istdev	Cmin	Cmax	Cmean	Cstdev	Arr.Rate	Samples
70	892	902	895.8	2.4	2630	4533	3720.6	572.5	6935.22	120
78	892	899	896.7	1.6	2560	3190	2758.6	82.1	6741.13	570
1532	892	899	896.9	1.5	2619	4149	2845.3	129.4	1142.38	570

Perhaps the most useful type of processed output is the **`Packet Table`**. This table summarizes instruction and cycle count data according to network packet size. All data collected for a particular packet size is summarized into a single row in the packet table. The columns of this table are arranged as follows:

`len Imin Imax Imean Istdev Cmin Cmax Cmean Cstdev Arr.Rate Samples`

`len`	The length of the frame. All log entries associated with the cost of processing a particular frame size will be summaried in one row.
`Imin`	The minimum number of instructions needed to process frames of this size across all logged frames
`Imax`	The maximum number of instructions needed to process frames of this size across all logged frames.
`Imean`	The average number of instructions needed to process frames of this size across all logged frames.
`Istddev`	The standard deviation of the number of instructions needed to process frames of this size across all logged frames.
`Cmin`	The minimum number of cycles needed to process frames of this size across all logged frames
`Cmax`	The maximum number of cycles needed to process frames of this size across all logged frames.
`Cmean`	The average number of cycles needed to process frames of this size across all logged frames.
`Cstddev`	The standard deviation of the number of cycles needed to process frames of

	this size across all logged frames.
Arr.Rate	Average packet inter-arrival time, in milliseconds. This can be useful in calculating the number of packets per-second that can be processed.
Samples	Total number of packet samples of this particular size that went into these calculations.

Porting

There are two key routines that need to be ported to any new platform in order to make performance logging work - `il_loginit()` and `il_log()`.

`il_loginit()` is used to perform any necessary processor-specific initialization. `il_log()` captures performance data and calls the common routine `ilc_log`.

x86

On x86 Pentium Pro platforms, `il_loginit()` writes a model-specific register to force the processor to keep track of instruction counts, cycle counts, or both. See *Pentium Pro Family Developer's Manual, Vol. 2* or any of the numerous other Intel performance analysis documentaion for details on the usage of the `wrmsr` instruction.

```
#define wrmsr(msr,val1,val2) \
     __asm__ __volatile__("wrmsr" \
                          : /* no outputs */ \
                          : "c" (msr), "a" (val1), "d" (val2))

static void
il_loginit(void)
{
#if defined(BCMDBG) && defined(__i386__)
    if (il_msg_level & 2048)
            wrmsr(0x186, 0x4700c0, 0);     /* enable P6 instr
counting */
#endif /* BCMDBG && __i386__ */
}
```

The other routine that must be ported to each platform is `il_log()`, which collected processor-specific performance data and calls the common log function `ilc_log()`. This version of `il_log()` for the Pentium uses the `rdtscl` and `rdmsr` instructions to read the cycle count and instructions count, respectively, each time `il_log()` is called.

```
/*
 * Obtaining cycle and instruction counts is processor specific.
 * Implement only the x86 version for now.
 */
void
il_log(char *fmt, ulong a1)
```

```
{
#ifdef __i386__
  static uint32 lasttsc = 0;
  static uint32 lastinstr = 0;
  uint32 tsc, instr, dummy;

  /* read cycle counter */
  rdtscl(tsc);

  /* read ctr1 value */
  rdmsr(0xc1, instr, dummy);     /* read P6 CTR0 register */

  /* call common log handler */
  ilc_log(tsc - lasttsc, instr - lastinstr, fmt, a1);

  /* update last values */
  lasttsc = tsc;
  lastinstr = instr;
#endif   /* __i386__ */
}
#endif   /* BCMDBG */
```

MIPS

On MIPS platforms, our `il_loginit` and `il_log` routines look like this,

```
void
il_loginit(void)
{
#if defined(BCMDBG)
  if (et_msg_level & 2048) {
          __u32 mode = 0x4f;            /* U,S,K,EXL,instr */
        __asm__ __volatile__(
                ".set\tnoat\n\t"
                "move\t$1,%0\n\t"
                ".word\t0x4081C800\n\t" /* MTPS: CPR[0,0] <-
GPR[1] */
                ".set\tat\n\t"
                :
                :"r" (mode)
                :"$1");
  }
#endif
}


/*
 * Obtaining cycle and instruction counts is processor specific.
 * Implement only the x86 version for now.
 */
void
il_log(char *fmt, ulong a1)
{
  static uint32 lasttsc = 0;
  static uint32 lastinstr = 0;
  uint32 tsc, instr, dummy;

  /* read cycle counter */
```

```
  tsc = read_32bit_cp0_register(CP0_COUNT);
  __asm__ __volatile__(
          ".set\tnoat\n\t"
          ".word\t0x4001C801\n\t"       /* MFPC: GPR[1] <-
CPR[0,0] */
          "move\t%0,$1\n\t"
          ".set\tat\n\t"
          :"=r" (instr)
          :
          :"$1");

  /* NEC VR5432 counts backwards */
  etc_log(tsc - lasttsc, lastinstr - instr, fmt, a1);

  /* update last values */
  lasttsc = tsc;
  lastinstr = instr;
}
```

5.4 Bring-up Considerations

Most times, verifying the functionality of the Broadcom BCM42xx component in your embedded design is as easy as compiling the driver for your platform and executing it. But there will be exceptions: perhaps the hardware is back before the driver is ready or the driver is ported but it isn't working as expected. This document is targeted to overcome such initial bring-up hurdles so that you can verify your hardware design and get your BCM42xx driver up and running quickly and smoothly.

Verifying BCM42xx Register Accesses

Many times, it is desirable to determine if the CPU in the design can communicate with the BCM42xx. Depending on the stage of the software development, there are a couple of ways to do this.

Verifying Register Access Pre-port

If you have not completed the port of the driver software, you can still verify BCM42xx connectivity via read and write to appropriate registers as follows. This pseudocode assumes that you have the following functions: ReadReg16(int offset) which reads a 16-bit register at the specified offset from the base address of the BCM42xx; ReadReg32(int offset) which reads a 32-bit register at the specified offset from the base address of the BCM42xx; WriteReg16(int offset, int value) which writes a 16-bit value to the register at the specified offset and WriteReg32(int offset, int value) which writes a 32-bit value to the register at the specified offset.

Perform a software reset on the BCM42xx:

```
        WriteReg32(0x00, 0x301);
        Delay(1);
        WriteReg32(0x00, 0x301);

        Verify 16-bit register access:
        x = ReadReg16(0x180);
        if (x != 0xFEDA)
        fail;

        Verify read of slow-access register.  If there are any errors,
         please consult the "Slow Access Registers" section for
remedies:
        WritReg16(0x40, 0xa55a);
        x = ReadReg16(0x40);
        if (x != 0xa55a)
                fail;

        Perform 32-bit write/read tests to check for sticky bits:
        WriteReg32(0x10, 0xa55aa55a);
        x = ReadReg(0x10);
        if (x != 0xa55aa55a)
                fail;

        WriteReg32(0x10, 0x5aa55aa5);
        x = ReadReg(0x10);
        if (x != 0x5aa55aa5)
                fail;
```

Verifying Access Post-port

After you have successfully compiled in the BCM4xx driver for your platform, the driver itself can perform the register access tests. There is a function in the source code called validate_chip_access(). This function performs the steps necessary to determine if the registers can be properly accessed. If there are any errors, the code will throw an ASSERT (soft exception).

Interrupts and Timers

After you can successfully access the registers, the last hurdle is often ensuring that the timers and interrupts in the device are firing properly.

Interrupts

Of course, proper generation of interrupts within the system is necessary. Be sure that you have made the appropriate RTOS system calls so that the ISR function is called whenever the BCM42xx generates an interrupt.

If you have another HPNA 2.0 device connected to your system, you should be receiving link packets once a second. If there is a link LED, that should remain on at this time indicating a successful link between the nodes. Ensure that the ISR is being called at this interval and that you are processing receive packets.

Timers

Equally important to proper operation of the BCM42xx driver are the timers. Various HPNA protocols (link, LARQ) rely on timers to keep proper state and generate the necessary control packets.

You can verify if the timers are functioning properly by ensuring that the il_watchdog() function is called once a second. This function is responsible, among other things, with generating link packets.

Enabling Debug Output

If the driver is still not functioning properly, you can enable debug output from the driver. This output is typically sent to the console and can be quite useful in determining if the driver is encountering error conditions, etc.

There is a variable in the ilc.c file called il_msg_level. By default (non debug mode), this value is set to 0 so no debug output is generated. Compile the code in debug mode (via -DDBG compile flag), change this value to 7 and run the code again. If you need assistance in understanding the debug output, please capture it to a file and send to hnsupport@broadcom.com for review.

Chapter 6: Using the SPROM

This section provides information on the operation and contents of the BCM42xx and BCM44xx InsideLine home networking controllers parameter SPROM (EEPROM or E2PROM) and gives the standard (typical) contents for each address location.

A serial EEPROM requires data to be input and output in a serial bit stream with a particular protocol, depending on its size and architecture or organization. The BCM42xx and BCM44xx devices use an internal state machine that avoids the problem of having to create and receive the required serial bit streams in the host software. To the host system CPU, the SPROM locations look just like 16-bit registers that can be read or written to in the normal way. On the BCM44xx family there is an SROM write enable bit (SPROMWriteEn in the Device Control Register) which needs to be set to one before writes to the SPROM will work. The BCM42xx chips do not implement this bit. After setting the write enable bit as needed, no special code is required to write to an SPROM location. This greatly simplifies the job of the software programmer.

The InsideLine chips support 1-Kbit, 4-Kbit, or 16-Kbit Microwire SPROM devices. The presence of a 10K Ω pull-down resistor on the SPROM_DIN signal (pin 51, which is sampled at power-on reset time) indicates that an SPROM is present. Certain embedded systems applications do not require an SPROM in

the design – in these cases the pull-down resistor should not be present. Embedded systems applications that require power management must use an SPROM with the BCM42xx/BCM44xx to set the Power Management Enable Bits in the PMCR – this register is read only from the host system bus side. All other functions are writeable through the host bus interface, so if power management is not required in an embedded system application, there is no need to have an attached SPROM – the host processor can write all the required registers at initialization time.

The normal SPROM size used is a 1-Kbit device (a 93C46 type device). For this size device, no other external strapping resistors are needed. If a 4-Kbit device is used (93C66), a 10KΩ pull-up resistor is required on the SPROM_DOUT signal (pin 52) to indicate the 4-Kbit size option to the BCM42xx/BCM44xx at power-on reset time. Likewise, for a 16-Kbit device (93C86), a 10KΩ pull-up resistor is required on the SPROM_CLK signal (pin 53) to indicate the 16-Kbit size. In the BCM4211 and all BCM44xx chips only, the values of SPROM_DOUT and SPROM_CLK ar reset time are available in the SPROMSize field of the Device Status Register, making it possible for the software to know the size of the SPROM. If no 10KΩ pull-down resistor is present on pin 51 (indicating no SPROM present), the pull-up resistors on pins 52 and 53 are ignored.

The Din pin of the SPROM should be connected to the BCM42xx/BCM44xx SPROM_DOUT pin and the SPROM's Dout pin should be connected to the BCM42xx/BCM44xx SPROM_DIN pin. All write accesses to the SPROM must be 16-bit aligned. Read accesses can be any size, but always returns a full 32 bits.

To eliminate an additional card hotswap, currently needed for the manufacturing test scripts for both the BCM42xx and BCM44xx, all SPROMs must be minimally pre-programmed before being installed on the PCB. For BCM4210 based designs, the SPROM must have the first word programmed to the value 0x0007 - the rest of the contents (0xFFFF) can be left alone. For BCM4211 based designs, the SPROM must have the first word programmed to the value 0x00A7 - the rest of the contents (0xFFFF) can be left alone. Pre-programming the SPROMs to these values allow the manufacturing bring-up/tests to flow smoothly without interruption. The manufacturing test software can program the rest of the SPROM contents normally.

The first 32 words (64 bytes) of the SPROM are reserved as the Hardware Configuration Area. The remaining SPROM memory space is available for other system or software uses. To ensure industry-wide compatibility, the extra space has been divided into three areas, each with some minimal level of definition. The figure below shows how the SPROM memory space is allocated.

The contents of the first 16 words of SPROM (offset 0x00 to 0x1E) are loaded into the BCM4210 and BCM4211 devices following the trailing edge of PCI_RESET. This action does not take place if an SPROM is not present. If no

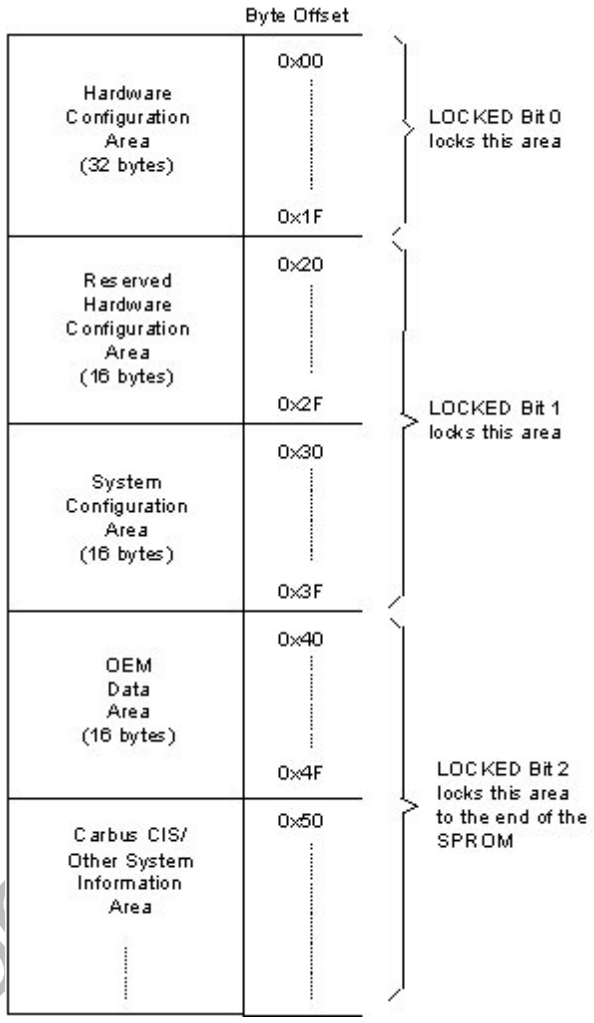
SPROM is present, all 16 words of hardware configuration take their power-on reset default values. The next block of eight words (offset 0x20 to 0x2E) is reserved for hardware configuration use and must be set to all zeros.

The next block of eight words (offset 0x30 to 0x3E) is defined as a System Configuration Area. This area contains one word for board revision, three words for a secondary MAC address and four words reserved for future system enhancements. The board revision word can contain any value. The Broadcom SPROM programming utility lets you set a major and a minor board revision level (for example, 1.2) that gets stored as two separate bytes at location 0x30 and 0x31. The secondary MAC address locations allow a vendor to store an alternative to the primary MAC. This can be useful for thid-party board vendors. The next seven bytes of the System Configuration area are reserved and are undefined. The last byte contains a checksum1 (CRC8) of the SPROM contents from byte offset 0x00 to 0x3F.

The next eight words (offset 0x40 to 0x4E) are defined as the OEM Data Area. Vendors can use this area for any purpose, such as, to store the board's serial number and date of manufacture.

The last area in the SPROM (from 0x50 to the end) is of variable length, depending on the size of the SPROM. This area is reserved for CardBus CIS information and other system specific uses. The last byte in the SPROM is reserved for a checksum1 (CRC8) of the data in this area (from 0x40 to the end).

The figure below also shows the relationship between the three SPROM lock bits and the areas in SPROM memory that they lock. Beware - after the LOCKED Bit 0 bit has been set, it is impossible to change the contents of the LOCKED Bits in the Hardware Configuration Area without removing the SPROM chip from the board and reprogramming it.



SPROM Memory Allocation

The table below shows the contents of the SPROM by address location, with the typical value for each location. In memory space, the SPROM's addresses start at an offset of 0x800 from the BCM42xx/BCM44xx registers. So, for example, if the BCM42xx/BCM44xx is on a PCI card at base address 0xF4201000, the first SPROM location would be at address 0xF4201800. Note that where three possible values are shown for a location, the first is for BCM4210 based designs, the second is for BCM4211 based designs; and the third is for BCM44xx based designs.

Typical SPROM Contents

SPROM Area	Address (Byte Offset)	Typical Contents	Parameter/s	Description
Hardware Configuration	0x00 0x01	0x0007/ 0x00A7/	SPROM Lock Bits	Hardware lock bits

SPROM Area	Address (Byte Offset)	Typical Contents	Parameter/s	Description
Area		0x1807		
	0x02 0x03	0xFC00/ 0x7C00/ 0x7F00	Power management enable bits, expansion ROM enable, size, mode and timing	Power management and expansion ROM control
	0x04 0x05	0x0101/ 0x0103/ 0x105	PCI Subsystem ID	Subsystem ID – vendor specific
	0x06 0x07	0xFEDA/ 0x14E4/ 0x14E4	PCI Subsystem Vendor ID (Epigram/Broadcom)	Subsystem Vendor ID – each vendor has a specific assigned value
	0x08 0x09	0xFE03	BCM42xx: AFE Revision ID/ Supported Bits Per Baud (BPB)	Attached BCM4100 chip revisionEnable bits for 2 – 8 BPB
	0x08 0x09	0xXXX	BCM44xx: Primary iLine MAC Address [15:0]	Vendor's iLine MAC address
	0x0A 0x0B	0xXXXX	BCM42xx: Primary iLine MAC Address [15:0]	Vendor's iLine MAC address
	0x0A 0x0B	0xXXXX	BCM44xx: Primary iLine MAC Address [31:16]	
	0x0C 0x0D	0xXXXX	BCM42xx: Primary iLine MAC Address [31:16]	
	0x0C 0x0D	0xXXXX	BCM44xx: Primary iLine MAC Address [47:32]	
	0x0E 0x0F	0xXXXX	BCM42xx: Primary iLine MAC Address [47:32]	
	0x0E 0x0F	0xXXXX	BCM44xx: Ethernet MAC Address [15:0]	Vendor's Ethernet MAC address
	0x10 – 0x2F	0x0000	BCM4210: Reserved for Hardware Use	Reserved – set to zero
	0x10 – 0x17	0x0000	BCM4211: Reserved for Hardware Use	Reserved – set to zero
	0x10 0x11	0xXXXX	BCM44xx: Ethernet MAC Address [31:16]	
	0x12 0x13	0xXXXX	BCM44xx: Ethernet MAC Address [47:32]	

SPROM Area	Address (Byte Offset)	Typical Contents	Parameter/s	Description
	0x14 – 0x1F	0x0000	BCM44xx: Reserved for Hardware Use	Reserved – set to zero
	0x18 0x19	0x4212	BCM4211: Function 1 Device ID	PCI Function 1 Device ID for BCM4211
	0x1A 0x1B	0x7C02	BCM4211: Function 1 Enable and Power Management Bits	Function 1 Enable and Power Management bits for BCM4211
	0x1C – 0x2F	0x0000	BCM4211: Reserved for Hardware Use	Reserved – set to zero
	0x20/ 0x21	0x7D00	BCM44xx: Codec Power Mgmt & AFE Rev	Codec (PCI function 1) PME & AFE Revision
	0x22/ 0x23	0x7C22	BCM44xx: Ethernet Control & Power Mgmt	Ethernet (PCI function 2) Enable/Link & PME
	0x24/ 0x25	0x0000	BCM44xx: Reserved for Hardware Use	Reserved – set to zero
	0x26/ 0x27	0x0000	BCM44xx: iLine D2/D1/D0 Power	Power settings for iLine states D2/D1/D0
	0x28/ 0x29	0x0000	BCM44xx: iLine D3, Codec D0 Power	Power settings for iLine state D3 and Codec state D0
	0x2A/ 0x2B	0x0000	BCM44xx: Codec D3/D2/D1 Power	Power settings for Codec states D3/D2/D1
	0x2C/ 0x2D	0x0000	BCM44xx: Ethernet D2/D1/D0 Power	Power settings for Ethernet states D2/D1/D0
	0x2E/ 0x2F	0x0000	BCM44xx: Ethernet D3 & Total Power	Power settings for Ethernet state D3 & total device power
System Configuration Area	0x30 0x31	0xXXXX	Board Rev – Minor Board Revision - Major	Board revision number (e.g., board revision 2.3 would result in 0x0203)

SPROM Area	Address (Byte Offset)	Typical Contents	Parameter/s	Description
	0x32 (Bits 7:0) 0x33 (Bits 15:8)	0xXXXX	Secondary MAC Address [15:0]	Secondary MAC Address (3rd party)
	0x34 (Bits 23:16) 0x35 (Bits 31:24)	0xXXXX	Secondary MAC Address [31:16]	
	0x36 (Bits 39:32) 0x37 (Bits 47:40)	0xXXXX	Secondary MAC Address [47:32]	
	0x38 – 0x3E	0x0000	Reserved for System Use	Reserved – set to zero
	0x3F	0xXX	Checksum #1	CRC8 checksum[1] of SPROM locations 0x00 through 0x3F
OEM Data Area	0x40 – 0x4F	0xXXXX	OEM data	Can be anything the vendor desires
Other System Information	0x50 - end	0xXXXX	Cardbus CIS Information/Other System Information	Can be anything the vendor desires
	Last byte	0xXX	Checksum #2	CRC8 checksum[1] of SPROM locations 0x40 through the last byte

The CRC8 checksums are computed using the polynomial:

```
x^8 + x^7 +x^6 + x^4 + x^2 + 1
```

E2PROM Programming Notes

- Because the SPROM is a serial device, it takes some time to read the data. The serial interface runs at just over 500 KHz clock rate, so you must allow 1 ms after a read or write to complete an event at the SPROM. You must have a delay loop built into the software to access the SPROM. Broadcom recommends a software wait time of about 1 ms before fetching the data. Do not hold up the CPU for this long while waiting for the data to be fetched. Most embedded CPUs only allow a certain number of wait states, typically four or eight.
- Some E2PROMs (for example, the AT93C46-10TC-2.7) do not have Schmitt inputs and are subject to random data corruption and erasure due to noise on the inputs during power-up and power-down events. There is a continuous return rate from the field if non-Schmitt input parts are used.
- Selection of the right SPROM device is essential. The SPROMs might not program properly if they are the wrong voltage. For example, the 93C14M8 parts that have the 4.5V to 5.5V Vdd die do NOT work reliably at 3.3V. You must either change the parts to 93C14LM8 (with the low voltage 2.7V die) or to the more commonly used 93C46LM8 parts.

Designing Without an SPROM

An SPROM is not *mandatory* for designing with the BCM4210. However, there must be a place to store the MAC addresses and any other pertinent information derived from the SPROM (for example, on-board FLASH memory).

If you decide not to use one, then you must do the following:

1. Ensure that there is no SPROM physically present (SPROM_DIN physically No Connect)
2. Compile with the `-DNOSPROM` option (in makefile, for example, BCM42xx.mk) e.g.
3. For iLine designs without an SPROM, the driver in the per-port code assigns a default MAC address (for example, in il_psos.c) with the following structure definition:

   ```
   static construct ether_addr ether_default = {{0, 1, 2, 3, 4, 0}}
   ```

   It then copies that into the following "cur_etheraddr" and "perm_etheraddr" variables, corresponding respectively to the local Ethernet address and the Ethernet address as read from the SPROM, if present.

   This is done in the following section of the per-port code:

   ```
   #ifdef NOSPROM
   bcopy(eaddr, &ilc->perm_etheraddr, ETHER_ADDR_LEN);
   bcopy(&ilc->perm_etheraddr, &ilc->cur_etheraddr,
ETHER_ADDR_LEN);
   ```

```
#endif

where
#define eaddr ((char *) &ether_default)
```

Ensure that the default MAC address assigned above as default by the driver is replaced by "per-port" code that reads the "TRUE" MAC address of the device under test. In the absence of an SPROM, this MAC address must be available to the driver during initial load from some alternate storage location in the system.

SPROM Utility [PCI NIC Only]

On request, Broadcom can provide a binary utility that is a WIN32 console mode application that can program the serial E2PROM attached to the BCM42xx. It uses a driver to access the chip directly. This utility can also dump the contents of the SPROM and it can check or update CRC in the serial E2PROM.

Appendix A: Frequently Asked Questions

Q) What is the maximum frame rate for iLine10?

A) The maximum frame rate for iLine10 is around 8000 frames/second using MINSIZE packets and 1173 fps using MAXSIZE packets.

Q) Is it necessary for iLine10 embedded drivers to disable the system interrupts using the perimeter locking mechanism?

A) Yes. Our driver design requires this locking mechanism. It synchronizes various sections of the driver in terms of resource sharing and state preservation. It also prevents re-entrancy.

Q) Can perimeter locking be implemented differently?

A) Probably, but it would involve very complex architecture that preserves states in every possible case where other interrupts can come in. Keeping a simplified perimeter locking mechanism ensures multiple entries into a process under execution and guarantees state preservation. For example, you do not want a watchdog interrupt happening in the middle of il_send(). One of the many critical tasks protected by this locking is message queuing and de-queueing. We took a simplified approach of disabling all interrupts. As long as we can ensure that no other entry point in the driver can fire before the current code completes the "locked" section, it should be fine.

Q) In the current iLine10 drivers, is it absolutely necessary for all frame buffers to be aligned on a 2K boundary?

A) DMA requires that they do not cross 4Kbyte boundaries. The 2 Kbyte buffers allow easy/efficient alloc/free process. Also, one cannot fit more than two full-size frames in 4 Kbytes anyway. If there is really a memory constraint, then you could play some tricks to squeeze the buffers (improve memory utilization).

Q) Do the frame buffers have to *begin* on 2K boundary?

A) No. For the receive path, our chip has no alignment or size restrictions except that buffers cannot cross 4 Kbytes boundaries (that is, they cannot be larger than 4 Kbytes). We optimize whenever possible to PCI burst boundaries (and avoid non-bursts when possible) and use the DMA rxoffset value to get the start of the ether_header aligned 2-mod-4 so that the data following the ether_header is at least four-byte aligned.

Q) What is the purpose of having a software copy of the interrupt mask in the driver?

A) The software intmask is an NDIS -ism to allow the dpc to reset and reinit the chip and not enable interrupts as a side effect -- think of it as write-through value that is what we intend the hardware intmask to be.

Q) Why are the fields of MSI interrupt status register set when the application is running in DMA mode? Can the MSI interrupt and mask registers be non-zero, even when not operating on MSI mode?

A) Some of the intstatus and msiintstatus signals are shared on the chip. So the answer is YES, both the MSI interrupt and mask registers can be non-zero even in non-MSI cases. If you are using PCI, ignore the msiintstatus/msiintmask. When you are using MSI, ignore the intstatus/intmask.

Q) Why am getting excessive collisions in my HPNA network?

A) Check the following:

1. Have your filters reviewed by hardware team (both yours and Broadcom's). It is possible that the filter diodes are hooked up the wrong way, causing excessive collisions.
2. The chip might have been initialized with an all-zero Ethernet MAC address.
3. Running with an un-terminated RJ-11 could result in false collisions. If you try to send packets from a port that has nothing connected to it, then the "sending station" can experience "false collisions." This is because, without some kind of termination, the signal reflected at the sending hybrid

is so large that it distorts in the analog portion of the sender's receive path. The collision detection algorithm in the BCM42xx falsely senses the distortion as a "collision."

4. Check the transmit path in the box hardware carefully. Broadcom uses "FCC part 68" signal level spec as a guideline (HPNA 2.0 spec): a maximum of -15 dBV (or 178 millivols rms) measured over a 2 µs window across a 135 ohm resistive termination.

Q) What is the difference between PROMISCUOUS and PROMISCTYPE mode in the driver?

A) In Promiscuous mode (which a bridge would be in), the driver receives and passes up all packets in the protocol stack except the ILCP frames.

In Promisctype mode, ALL frames, including out of band ILCP iLine Control packets like LARQ, RATE, and CSA, are passed up. This is not the normal operating mode and the driver must be explicitly put in this mode. Promisctype mode is used mainly for certification testing.

Q) How does LARQ work in both Transmit and Receive side?

A) TRANSMIT SIDE - When a packet is transmitted, LARQ duplicates the packet, adds the duplicate to the channel's save queue, and starts the save timer. As long as the packet is in the save queue, it is available for retransmission. A packet is removed from the save queue when the save timer expires, the channel's save queue exceeds the per-channel limit, or the system's save queues exceed the global limit and the packet is the globally oldest saved packet.

RECEIVE SIDE - When LARQ notices that a packet is missing (either received in error or not recevied), it creates an entry on the channel's hold queue and starts the hold timer. Any packets subsequently received on that channel are added to the hold queue. When the missing packet is re-transmitted and successfully received, it and the subsequently received packets are indicated up. If the hold timer fires before the packet is received, the packet is declared lost and the subsequently received packets are indicated up. In all cases, buffers are eventually returned to the pool.

Q) What are the OSs currently supporting iLine10 PVS test for the Controller and Reference machines?

A) Windows NT 4.0 SP5 or better. It also runs on Windows 2000. Currently, there is no support for Windows 98, or for Windows 95 and Windows ME.

Q) Are there special conditions or timing issues to write to SPROM configuration area in an iLine10 design?

A) Any serial access to the SPROM takes some time to complete. The serial interface runs at just over 500 KHz clock rate, so you must allow 20 ms between doing a read or write to the actual event being completed out at the SPROM device. Reads, on the other hand, can be executed back-to-back. Also, you must make sure that the LOCK bits in the configuration area are NOT set in word 00h.

Q) We currently develop our code under UNIX, and using C/C++ Compiler from Diab Data, Inc. Is there any compiler dependent issues that I should be aware of?

A) There is no known problem. We have used it here at Broadcom.

Q) While trying to compile pSOS driver from Broadcom, I got a compiler error, saying that it cannot find the bsp.h file. Where can I find this file?

A) The "bsp.h" comes with pSOS, not in the iLine10 driver from Broadcom.

Q) Can I use the general purpose timer (GPTimer) in the BCM42xx chip for other purposes?

A) The GPTimer in BCM4210 is used by the driver (in the CERT layer of the code) and using it for any other purpose is not recommended.

Q) How do you use the Logging function for Broadcom's new iLine10 drivers?

A) Refer to the IL_LOG Performance Measurement Tool section above for detailed information about using the Logging function.

Q) What might cause the performance to drop when I upgrade to the latest driver?

A) Use the il_dump() function to output the error counters on a terminal if you can. These can help determine whether there are any reported errors on the transmitter or receiver for the test you are running. Send Broadcom the log for analysis. Also, try running "netstat -s" (or appropriate TCP/IP tool) reports on any IP or TCP transmit or receive errors. Also check for TCP retransmissions.

Appendix B: Feedback and Links

We are interested in your feedback on this document. Please send suggestions including the text that you'd like to see changed/added to hnsupport@broadcom.com.

Links

www.broadcom.com

www.homepna.org