AN 12.12

LAN9118 Family Programmer Reference Guide

1 Introduction

This application note describes how to successfully develop a network device driver for LAN9118 Family products. It covers device features, software design techniques, network operating system interfaces, and critical design points. It provides basic design guidelines for incorporating LAN9118 Family products into networked applications. It overviews topics as microprocessor-LAN hardware interactions; initialization, interrupts, operating system, and communication protocol stack considerations.It also reviews good design practice for engineers getting acquainted with network device design.

1.1 References

This manual references the following documents:

- SMSC LAN9118 Datasheet
- SMSC LAN9117 Datasheet
- SMSC LAN9116 Datasheet
- SMSC LAN9115 Datasheet
- SMSC LAN9118 Reference Design Schematic
- SMSC LAN9117 Reference Design Schematic
- SMSC LAN9116 Reference Design Schematic
- SMSC LAN9115 Reference Design Schematic

Always refer to these documents for complete and current device information. Circuit examples shown in this document are for illustration only. Follow the corresponding Reference Design Schematic when implementing an actual circuit design of a LAN9118 Family device.

Please visit SMSC's website at http://www.smsc.com for the latest updated documentation.

1.2 Document Conventions

- In this document, the terms device, network device, controller, and network controller all refer to a controller in the LAN9118 Family, which includes the LAN9118, LAN9117, LAN9116 and LAN9115.
- Host refers to the system into which the device is designed, including the processor, and application software, etc.
- MAC stands for Media Access Controller; the portion of the device responsible for sending or receiving blocks of data from the network.
- A packet is a complete Ethernet frame not yet sent through the MAC, or a frame after it has been received by the MAC.
- A frame is a complete Ethernet data frame with the 802.3 Layer 2 (**MAC**) header included.
- A bit is a single binary digit that has a value of 1 (high) or 0 (low). A field is a continuous string of bits of whatever length specified.

- An octet or byte is a group of 8 bits, treated as a single unit, with a value of 0-255 unsigned, or -127 to +127 signed. A byte is generally the smallest unit of data that can be individually addressed.
- A word or short int is a group of 16 bits or two bytes,2 adjacent **bytes**, representing a 16-bit, single symbol or a numeric range from 0 – 65,535 unsigned, or +/- 32,767 as a signed value. **WORD** values are aligned on 2-byte memory boundaries. Their addresses are always expressed in even number terms ending in 0x0, 0x2, 0x4, 0x6, 0x8, 0xa, 0xc, and 0xe.
- A DWORD or long int always refers to 4 adjacent bytes, representing a 32-bit, single symbol or a numeric range from 0 – 4,294,967,295, or +/- 2,147,483,647 as a signed value. **DWORD** values are aligned on 4-byte memory boundaries. Their addresses are always expressed in even number terms ending in 0x0, 0x4, 0x8, and 0xc.

2 Controller Overview

LAN9118 Family devices are full-featured, single-chip 10/100 Ethernet controllers designed for embedded applications where performance, flexibility, ease of integration and low cost are required. LAN9118 Family devices are fully IEEE 802.3 10BASE-T and 802.3u 100BASE-TX compliant.

LAN9118 Family devices include an integrated Ethernet MAC and PHY with a high-performance SRAM-like slave interface. The simple, yet highly functional host bus interface provides glue-less connection to most common 32- and 16-bit microprocessors and microcontrollers, including those 32-bit microprocessors presenting a 16-bit data bus interface to the device. LAN9118 Family Devices include large transmit and receive data FIFOs with a high-speed host bus interface to accommodate high bandwidth, high latency applications. In addition, the devices memory buffer architecture allows the most efficient use of memory resources by optimizing packet granularity.

2.1 Block Diagrams

2.1.1 Internal Block Diagram

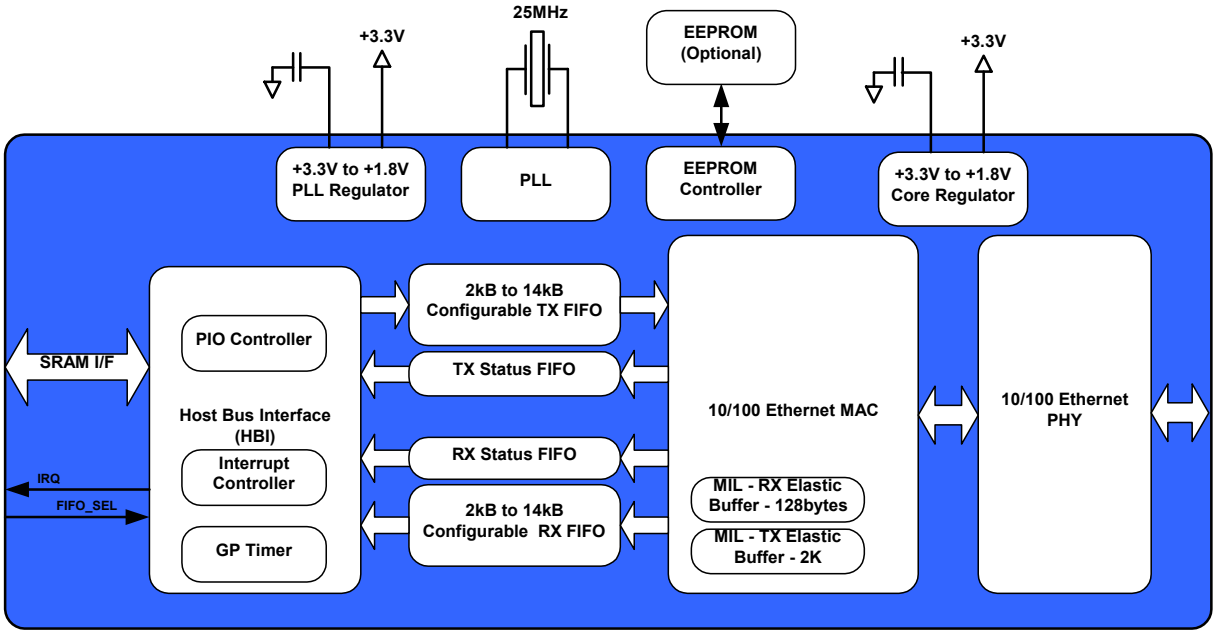

Figure 2.1 LAN9118 Family Device Internal Block Diagram

2.1.2 System Level Block Diagram

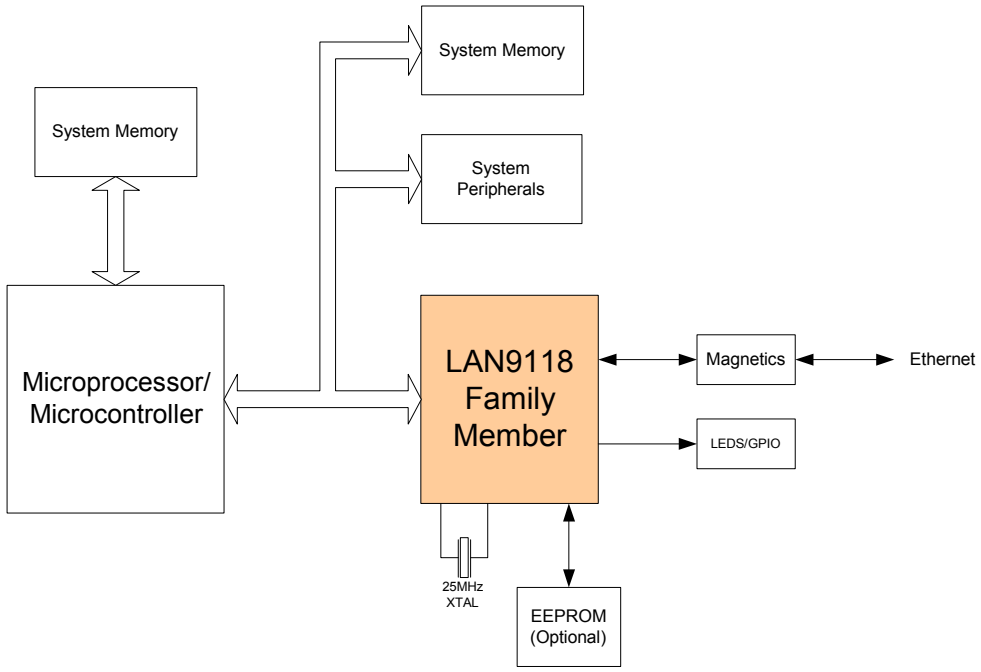

Figure 2.2 LAN9118 Family Device System-Level Block-Diagram

2.2 Common Product Family Features

- Single chip Ethernet controller
  Fully compliant with IEEE 802.3/802.3u standards
  Integrated Ethernet MAC and PHY
  10BASE-T and 100BASE-TX support
  Full- and Half-duplex support
  Full-duplex flow control
  Backpressure for half-duplex flow control
  Preamble generation and removal
  Automatic 32-bit CRC generation and checking
  Automatic payload padding and pad removal
  Loop-back modes
- Flexible address filtering modes
  One 48-bit perfect address
  64 hash-filtered multicast addresses
  Pass all multicast
  Promiscuous mode
  Inverse filtering
  Pass all incoming with status report
  Disable reception of broadcast packets
- Integrated Ethernet PHY
  Auto-negotiation
  Automatic polarity correction

- High-Performance host bus interface
  - Simple SRAM-like interface
  - Large, 16Kbyte FIFO memory with adjustable Tx/Rx allocation
  - Memory Alignment Technology (MAT) supports interleaved transmit/receive/command/status access
  - One configurable Host interrupt
  - Burst read support
- Comprehensive power management features
  - Numerous power management modes
  - Wake on LAN
  - “Packet-of-Interest” wakeup
  - Wakeup indicator event signal
  - Link Status Change
- Miscellaneous features
  - Low profile 100-pin TQFP package
  - Single 3.3V power supply with 5V tolerant I/O
  - General Purpose Timer
  - Support for optional serial EEPROM
  - Supports for 3 LEDs/GPIO signals

3 Register Description

Refer to the LAN9118 Family datasheets for complete descriptions of the Control and Status Registers (CSRs), as well as for descriptions of register and bit names, nomenclature and attributes used in this application note. Highlights are reproduced here for quick reference.

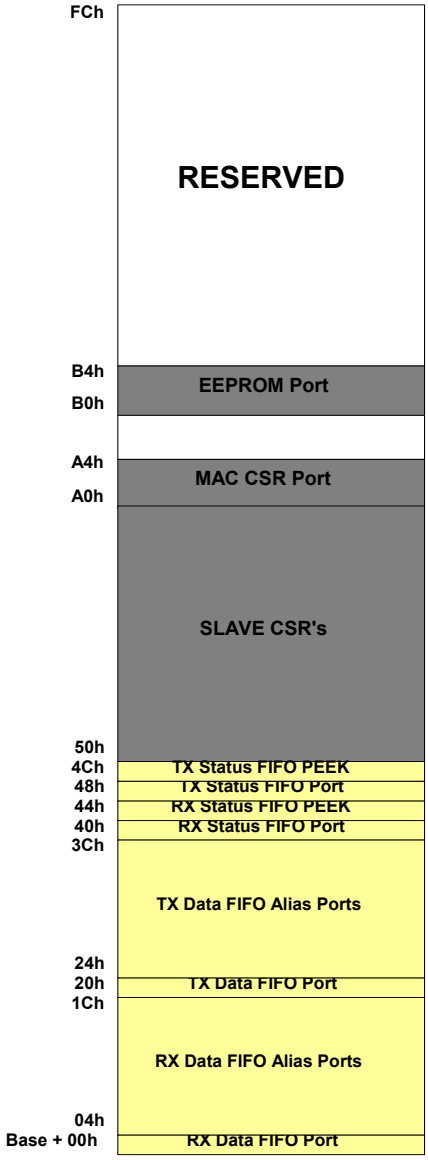

Figure 3.1 LAN9118 Family Device Memory MAP

3.1 Directly Addressable Registers

These registers are also referred to as “Slave Registers”.

Table 3.1 LAN9118 Family Directly Addressable Register Map

OFFSET	SYMBOL	REGISTER NAME	DEFAULT
50h	ID_REV	Chip IP and Rev	01180001h
54h	INT_CFG	Main Interrupt Configuration	00000000h
58h	INT_STS	Interrupt Status	00000000h
5Ch	INT_EN	Interrupt Enable Register	00000000h
60h	RESERVED	Reserved for future use	-
64h	BYTE_TEST	Read-only byte order testing register	87654321h
68h	FIFO_INT	FIFO Level Interrupts	48000000h
6Ch	RX_CFG	Receive Configuration	00000000h
70h	TX_CFG	Transmit Configuration	00000000h
74h	HW_CFG	Hardware Configuration	00000800h
78h	RX_DP_CTRL	RX Datapath Control	00000000h
7Ch	RX_FIFO_INF	Receive FIFO Information	00000000h
80h	TX_FIFO_INF	Transmit FIFO Information	00001200h
84h	PMT_CTRL	Power Management Control	00000000h
88h	GPIO_CFG	General Purpose IO Configuration	00000000h
8Ch	GPT_CFG	General Purpose Timer	0000FFFFh
90h	GPT_CNT	General Purpose Timer Count	0000FFFFh
94h	RESERVED	Reserved for future use	-
98h	WORD_SWAP	WORD_SWAP	00000000h
9Ch	FREE_RUN	Free Run Counter	-
A0h	RX_DROP	RX Dropped Frames Counter	00000000h
A4h	MAC_CSR_CMD	These two registers are used to access the MAC CSRs.	00000000h
A8h	MAC_CSR_DATA		00000000h
ACh	AFC_CFG	Automatic Flow Control	00000000h
B0h	E2P_CMD	These two registers are used to access the EEPROM	00000000h
B4h	E2P_DATA		00000000h
B8h - FCh	RESERVED	Reserved for future use	-

3.2 MAC Control and Status Registers

The registers listed below are accessed indirectly through the MAC_CSR_CMD and MAC_CSR_DATA Registers. These registers are used in Section 5.1.2 and Section 5.5.

.

Table 3.2 LAN9118 Family MAC CSR Register Map

MAC CONTROL AND STATUS REGISTERS			
INDEX	**SYMBOL**	**REGISTER NAME**	**DEFAULT**
1	MAC_CR	MAC Control Register	00040000h
2	ADDRH	MAC Address High	0000FFFFh
3	ADDRL	MAC Address Low	FFFFFFFFh
4	HASHH	Multicast Hash Table High	00000000h
5	HASHL	Multicast Hash Table Low	00000000h
6	MII_ACC	MII Access	00000000h
7	MII_DATA	MII Data	00000000h
8	FLOW	Flow Control	00000000h
9	VLAN1	VLAN1 Tag	00000000h
10	VLAN2	VLAN2 Tag	00000000h
11	WUFF	Wake-up Frame Filter	00000000h
12	WUCSR	Wake-up Status and Control	00000000h

3.3 PHY Registers

The PHY registers are accessed through two levels of indirection: through **MAC_CSR_CMD/DATA** Registers and the **MII_ACCESS/DATA** Registers. The PHY provides its own interrupt source and mask register; a “master” enable/disable bit for the PHY interrupts is found in the **PHY_INT_EN** bit in the **INT_STS/INT_EN** registers.

Individual PHY Registers are identified through an index field located in the **MII_ACC** register . PHY Register Indices are shown in Table 3.3 below. These registers are used in Section 5.6.

Note: PHY Register bits designated as NASR are reset when the SIM CSR Software Reset is generated. The NASR designation is only applicable when bit 15 of the PHY Basic Control Register (Reset) is set.

Table 3.3 LAN9118 Family PHY Control and Status Register

PHY CONTROL AND STATUS REGISTERS	
INDEX (IN DECIMAL)	**REGISTER NAME**
0	Basic Control Register
1	Basic Status Register
2	PHY Identifier 1
3	PHY Identifier 2
4	Auto-Negotiation Advertisement Register
5	Auto-Negotiation Link Partner Ability Register
6	Auto-Negotiation Expansion Register
17	Mode Control/Status Register
29	Interrupt Source Register
30	Interrupt Mask Register
31	PHY Special Control/Status Register

3.4 Restrictions on Read-Follow-Write CSR Accesses

There are timing restrictions on successive operations to some CSRs. These restrictions come into play whenever a write operation to a control register is followed by a read operation from a related register. These restrictions arise because of internal delays between write operations and their effects. For example, when the **TX Data FIFO** is written, there is a delay of up to 135ns before the **TX_FIFO_INF** register changes.

In order to prevent the host from reading invalid status, minimum wait periods have been established following write operations to each CSR. These periods are specified in Table 3.4 below. For each CSR, the host is required to wait the specified period of time after a write before performing any read. These wait periods are for read operations that immediately follow any write cycle. Note that the required wait period is dependant upon the register being read after the write.

Performing “dummy” reads of the **BYTE_TEST** register is a convenient way to guarantee that the minimum write-to-read timing restriction is met. Table 3.4 below also shows the number of dummy reads that are required before reading the register indicated. The number of BYTE_TEST reads in this table is based on the minimum timing for Tcyc (45ns). For microprocessors with slower busses, the number of reads may be reduced as long as the total time is equal to, or greater than the time specified in Table 3.4. Note that dummy reads of the **BYTE_TEST** register are not required as long as the minimum time period is met.

Table 3.4 Read after Write Timing Rules

REGISTER NAME	MINIMUM WAIT AFTER ANY WRITE CYCLE (IN NS)	NUMBER OF BYTE_TEST READS (ASSUMING 45NS TCYC)
ID_REV	0	0
INT_CFG	135	3
INT_STS	90	2
INT_EN	45	1
BYTE_TEST	0	0
FIFO_INT	45	1
RX_CFG	45	1
TX_CFG	45	1
HW_CFG	45	1
RX_DP_CTRL	45	1
RX_FIFO_INF	0	0
TX_FIFO_INF	135	3
PMT_CTRL	513	7
GPIO_CFG	45	1
GPT_CFG	45	1
GPT_CNT	135	3
WORD_SWAP	45	1
FREE_RUN	180	4
RX_DROP	0	0
MAC_CSR_CMD	45	1
MAC_CSR_DATA	45	1
AFC_CFG	45	1
E2P_CMD	45	1
E2P_DATA	45	1

3.5 Restrictions on Read-Follow-Read CSR Accesses

There are also restrictions on certain CSR read operations following other read operations. These restrictions arise when a read operation has a side-effect that affects another read operation. In many cases there is a delay between reading the a CSR and the subsequent side-effect. To prevent the host from reading invalid status, minimum wait periods have been established. These wait periods are implemented by having the host perform “dummy” reads of the BYTE_TEST register. The required dummy reads are listed in the Table 3.5 below:

Table 3.5 Special Back-to-Back Cycles

AFTER READING...	PERFORM X READS OF BYTE_TEST	BEFORE READING...
RX Data FIFO	3	RX_FIFO_INF
RX Status FIFO	3	RX_FIFO_INF
TX Status FIFO	3	TX_FIFO_INF
RX_DROP	4	RX_DROP

4 Programming Recommendations

The fundamental operations of the driver are initialization, transmit packet processing, receive packet processing, and interrupt processing. In addition, the driver needs to manage link negotiation and interact with the operating system. The driver may be required to keep statistics, manage power consumption, and control other functions.

4.1 The Necessity of Register Ownership

Writing a driver is a complex task. Every possible sequence of operations needs to be considered. A commonly encountered problem is having to share a resource, such as a CSR, between two or more code threads. Having multiple code threads access the same resource only increases the challenge. If one thread is interrupted by a second thread, such that both threads are attempting to modify the resource at the same time, unpredictable behavior may result. Collectively, these conflicts are known as synchronization issues.

4.2 The Importance of Planning (strcmp(“Fail to Plan”) == strcmp (“Plan to Fail”))

To prevent this type of unpredictable behavior, the first step in writing a driver is to plan which portions of the driver will access which registers. Every potential synchronization issue needs to be considered before the first line of code is written. Failure to do so will result in errors that are intermittent and extremely difficult to reproduce. Even worse, these problems may not even surface until well into the driver development. Initially, the driver may appear to work, but as integration proceeds and more threads are added, and the environment becomes more complex, the driver may suddenly and inexplicably develop problems.

4.3 Orthogonal Register Set

In order to prevent these types of synchronization issues, the register set of every LAN9118 Family device has been organized so that each register only needs to be accessed by a single thread. Registers which are accessed by the receive process do not need to be accessed by the transmit or initialization processes, and so on. In addition, registers used to manage flow control and separated from those used to manage the link.

The design of the register set, which is “orthogonal” allows the multiple threads within the driver to be interweaved, or run concurrently without interfering with each other.

4.4 Register Functionality in the LAN9118 Family Devices

Table 4.1, "Independent Data Threads Register Usage" details the functions of the most important LAN9118 Family CSRs

Table 4.1 Independent Data Threads Register Usage

RECEIVE REGISTER NAME	OFFSET	PURPOSE
RX_FIFO_INF	0x7c	Determine whether a packet is available
RX_STATUS_FIFO_PORT	0x40	Determine receive packet properties, such as size
RX_DATA_FIFO_PORT	0x00	Read received packet from device
TRANSMIT REGISTER NAME	**OFFSET**	**PURPOSE**
TX_FIFO_INF	0x80	Determine remaining free space in transmit FIFO, and number of packet statuses available
TX_STATUS_FIFO_PORT	0x48	Determine whether packets were sent successfully
TX_DATA_FIFO_PORT	0x20	Write packet data to device
FILTER REGISTER NAME	**OFFSET**	**PURPOSE**
MAC_CSR_CMD	0xa4	Access to **MAC_CR**, **HASHH**, and **HASHL**
MAC_CSR_DATA	0xa8	Access to **MAC_CR**, **HASHH**, and **HASHL**
LINK REGISTER NAME	**OFFSET**	**PURPOSE**
MAC_CSR_CMD	0xa4	Access to **MII_ACC** and **MII_DATA**
MAC_CSR_DATA	0xa8	Access to **MII_ACC** and **MII_DATA**
MII_ACC	0x6	Determine remaining free space in transmit FIFO, and number of packet statuses available
MII_DATA	0x7	Determine whether packets were sent successfully

Note: Notice that both the filter and link management functions depend upon the **MAC_CSR_** registers. This shows the importance of synchronizing access to these particular registers between the filter and link management threads.

4.5 An Example of Concurrency

The need of maintaining the independence of the transmit and receive packet processing functions has already been mentioned. For example, suppose that the transmit thread needs to execute operations T0-T9 below as part of transmitting a packet:

TIME 0, T+1, T+2, T+3, … T+19									
T0	T1	T2	T3	T4	T5	T6	T7	T8	T9

At the same time the receive thread needs to execute operations R0-R9 as part of receiving a packet:

TIME 0, T+1, T+2, T+3, … T+19									
R0	R1	R2	R3	R4	R5	R6	R7	R8	R9

It is entirely possible for these operations to be interleaved as follows:

TIME 0, T+1, T+2, T+3, … T+19																		
T0	**T1**	**R0**	**T2**	**R1**	**R2**	**T3**	**R3**	**R4**	**T4**	**T5**	**T6**	**T7**	**R5**	**R6**	**T8**	**R7**	**R8**	**R9**	**T9**

The only requirement is that within each thread, proper order of the data stream must be maintained; Receive operations must remain properly ordered and transmit operations likewise must remain properly ordered. Aside from the required ordering within each data stream, the streams can be interweaved arbitrarily; the controller must be capable of handling any legitimate possibility.

Interleaving also applies to the host interface. If both transmit and receive DMA channels are used, the transmit DMA channel can be writing a packet while the receive DMA channel is concurrently reading a packet. LAN9118 Family devices have been designed specifically to handle these kinds of cases.

Interleaved accesses also support interleaving between the link and filter management threads concurrent with the transmit and receive threads already described. This could occur if the transmit and receive threads were both involved in DMA activities, while the CPU was handling the filter thread at the moment a link-state change interrupted the processor. Because of the orthogonality of the register set, all four threads could be interleaved successfully without errors.

4.6 Software Interrupt Feature (SwInt)

LAN9118 Family devices provide a software interrupt feature (Swint). This feature allows the driver to force the device to activate its IRQ signal. This is done by setting the **INT_EN:SW_INT_EN** bit. One use of this feature is to allow the driver to verify that it has properly registered the Interrupt Service Routine to the correct IRQ during initialization.

Another use of this feature is to protect critical registers. One case where this feature can be used is when an ISR accesses a register and it is critical that this register not be accessed outside the ISR. If a code segment outside the ISR needs to access this register, rather than access the register directly, the segment can set a flag bit indicating that the register should be accessed and then invoke the ISR via the Swint feature. When the ISR sees the flag bit, it performs the register access on behalf of the code segment. The thread which called the ISR would also need some mechanism for determining that the request had completed, such as polling a status bit from the ISR.

Of course, this only works if there is only one segment of code outside the ISR needing to access the critical register. If there are multiple threads accessing the register, they need additional synchronization between them. Otherwise the activity of one thread may undo another; and if multiple threads call the ISR using the same flag bit, the ISR will not be able to tell which thread called it.

The first example works because of the assumption that only one thread will ever write to the flag and signal the SwInt. The SwInt handler can use the flag to interpret the requested operation. This is known as a producer-consumer relationship. That is, it is safe for multiple threads to share a variable if only one thread is the writer, or producer of the variable, and all other threads are readers, or consumers of that variable.

4.7 Ownership Policy in the Simple Driver

As a convenience to its customers, SMSC provides a simple driver for LAN9118 Family members, written to run under the Linux, which has been reduced in complexity and function in order to highlight the function of the device over the capabilities of a Linux device driver. Questions regarding register usage, synchronization and ownership can generally be resolved through this reference work. Table 4.2 below details the register ownership policy in the simple driver.

Table 4.2 Register Ownership Policy in the Simple Driver

REGISTER NAME	OWNERSHIP POLICY
RX_DATA_FIFO	Only used by the Rx thread, Rx_ProcessPackets
TX_DATA_FIFO	Only used by the TX thread, Tx_SendSkb
RX_STATUS_FIFO	Only used by the Rx thread, Rx_ProcessPackets
RX_STATUS_FIFO_PEEK	Not Used
TX_STATUS_FIFO	Used in Tx_CompleteTx in Tx_UpdateTxCounters. Tx_UpdateTxCounters is called by Tx_SendSkb in Simp911x_hard_start_xmit. Tx_UpdateTxCounters is also called by Simp911x_stop but only after disabling the TX queue in a multithreaded safe manner using the xmit_lock
TX_STATUS_FIFO_PEEK	Not Used.
ID_REV	Read Only
INT_CFG	Set in Lan_Initialize, Read in Simp911x_ISR 1) ClrBits in Simp911x_ISR 2) ClrBits in Rx_HandleInterrupt 3) SetBits in Rx_PopRxStatus 4) SetBits in Rx_ProcessPacketsTasklet Items 1, 2, 3, 4 are not in contention because 1 and 2, are part of the ISR, and 3 and 4 is the tasklet which is only called while the ISR is disabled.
INT_STS	Sharable
INT_EN	Initialized at startup. Used in Simp911x_stop
BYTE_TEST	Read Only
FIFO_INT	Initialized at start up. During run time only accessed by Tx_HandleInterrupt, and Tx_SendSkb and done in a safe manner.
RX_CFG	Only used during initialization
TX_CFG	Only used during initialization
HW_CFG	Only used during initialization
RX_DP_CTRL	Only used in Rx thread, in Rx_FastForward
RX_FIFO_INF	Read Only. Only used in Rx Thread, in Rx_PopRxStatus

Table 4.2 Register Ownership Policy in the Simple Driver (continued)

REGISTER NAME	OWNERSHIP POLICY
TX_FIFO_INF	Read Only. Used in TX thread, in Tx_GetTxStatusCount, Tx_SendSkb, and Tx_CompleteTx
PMT_CTRL	Only used during initialization, in Phy_Initialize
GPIO_CFG	Only used during initialization, in Lan_Initialize
GPT_CFG	Not Used
GPT_CNT	Not Used
WORD_SWAP	Not Used
FREE_RUN	Not Used
RX_DROP	Only used in Rx thread, in Rx_ProcessPackets
MAC_CSR_CMD	Protected by MacPhyAccessLock Except during initialization where only one thread is running.
MAC_CSR_DATA	Protected by MacPhyAccessLock Except during initialization where only one thread is running.
All **MAC** and **PHY** registers	Protected by MacPhyAccessLock Because MAC_CSR_ registers are protected, then all MAC and PHY registers are protected as well since they are access through the MAC_CSR_ registers.
AFC_CFG	Used during initialization, in Lan_Initialize. Also used during run time in timer call back, in Phy_UpdateLinkMode
E2P_CMD	Used during initialization, in Lan_Initialize
E2P_DATA	Not Used

5 Initialization

5.1 MAC Initialization

The **ID_REV** register is a good starting point from which to begin initialization, in that it provides a known location in memory containing a known value; in the case of a LAN9118, revision B, the value is 0x1180001. Typical usage is twofold. One is as a probe point, while the other use is as a discriminator of the LAN9118 Family member. The designation of the chip (0x0118/0x0117/0x0116/0x0115) is found in the upper 16-bits of the register, while the step revision is obtained from the lower 16-bits.

The **BYTE_TEST** register is provided to confirm the byte ordering of the host-device interface. The interface between the host and the device is designed correctly when the host can read this register as having a value of 0x87654321.

5.1.1 Software Reset of the MAC

Before performing a software reset operation to the MAC, the driver should ensure that the internal PHY is running. This can be determined by examining the **PMT_CTL:PM_MODE** field for any power-down state (non-zero). If this is found to be the case, a write to the **BYTE_TEST** register (a non-destructive write to a read-only register), followed by polling the **PMT_CTL:READY** bit until clear, will guarantee the PHY to be active and the initialization ready to continue.

When the driver is aware of the device, a software reset should be performed by writing a '1' value to the **HW_CFG:HW_CFG_SRST** bit. This bit is self clearing and can be polled to determine when to continue. Having finished the reset, the driver may then set the transmit FIFO size to a default value of 5 KB (**HW_CFG:TX_FIF_SZ** field, which is 5 KB after reset).

5.1.2 FIFO Allocation and Flow-Control Configuration

Then set the Automatic Flow Control Configuration (**AFC_CFG**) register High Level to a value of 7 KB (110, or 0x006e) and the Low Level control to a value of 3.5 KB (55, or 0x37), which breaks the FIFO up roughly in thirds. Also set the Backpressure duration to 50us for 100m operation (0x4), and 500 us for 10m operation.

5.1.3 Setting the MAC Address

5.1.3.1 The Unicast Address Register

LAN9118 Family members offer one 48-bit perfect (exact) address, whose value is divided between the MAC **ADDRH** and **ADDRL** registers. This address identities the controller to the LAN on which it is attached.

5.1.3.2 How a MAC Address is Stored

As an example, suppose the MAC address source address is supposed to be "**1a:2b:3c:4d:5e:6f**", where "**1a**" is the first octet sent on the wire. To do this, write the portion "**6f5e**" as the low 16-bits of the **ADDRH** register value and write the address portion "**4d3c2b1a**" to the **ADDRL** register.

5.1.3.3 Reloading the MAC Address from EEPROM

When a software reset operation is performed, it also starts the EEPROM reload cycle. To ensure that the EEPROM has finished loading, poll the EEPROM Command register (**E2P_CMD**) busy bit until clear. The loading should complete in under a millisecond. Upon completion, the **MAC_Address_Loaded** bit in the command register can be examined for success. This step should be completed before the driver sets up the General Purpose I/O configuration (GPIO_CFG), since some I/O pins are shared between EEPROM and GPIO usage.

Besides the MAC address, **no** other values are automatically loaded to the controller from EEPROM in a reload cycle.

5.1.3.4 Saving the MAC Address to EEPROM

A likely scenario during the manufacturing phase is to fix the MAC address of the host in the EEPROM after assembly. There may also be valid reasons where the driver would allow a user to change the MAC address after manufacture or startup. To reprogram the EEPROM under the control of the controller, first ensure that the address in the MAC ADDRx register is correct.

Then write the signature value of 0xa5 to the first location in EEPROM, followed by the MAC address as shown in Table 5.1 below:

:

Table 5.1 EEPROM MAC Address Layout

ADDRESS	CONTENTS
0	0xa5
1	0x1a
2	0x2b
3	0x3c
4	0x4d
5	0x5e
6	0x6f

5.2 Configuring Interrupts

All LAN9118 Family members provide an interrupt request signal (IRQ), which can easily be programmed for a variety of hardware environments.

5.2.1 Configuring the IRQ Pin

The polarity of the IRQ pin can be programmed to be active low or active high by setting the **IRQ_POL** bit in the **INT_CFG** register (1=high, 0=low). This programmability enables the device to accommodate a variety of processors with minimal external support circuitry.

The buffer-type of the IRQ pin can also be programmed to be either Open-Drain or Push-Pull by setting the **IRQ_TYPE** bit in the **INT_CFG** register (1 = Push-pull, 0 = Open Drain). If the Open-Drain setting is chosen, the polarity defaults to active low, and the setting of the **IRQ_POL** bit is ignored.

Once triggered, the IRQ output remains in an active state until acknowledged by the host.

5.2.2 Setting The Interrupt De-assertion Period

The interrupt de-assertion timer sets a minimum interval between assertions of the interrupt pin, allowing the host to control the percentage of time it dedicates to servicing the device, even during periods of heavy network traffic. The field controlling this timer (**INT_DEAS)** is located in the upper 8-bits field of the main interrupt configuration register (**INT_CFG).** This timer counts in 10us increments. The correct value for this timer depends on the processor, operating system and application. The Linux driver maintained on the SMSC website uses a de-assertion period of 220us (22d).

5.2.3 Enabling and Disabling Interrupts

Individual interrupts are controlled through the interrupt enable register (**INT_EN**) via read-modify-write operations. First read this register. To enable specific interrupts, OR the contents with a bit-mask which

has the bits corresponding with the interrupts to be enabled set to “1”. To disable specific interrupts, AND the contents with a bit-mask which has the bits corresponding to the interrupts to be disabled set to “0”. Write the modified value back to the register.

Per Section 3.4 and Section 3.5, the **INT_EN** register requires a “settling” time before its effects are felt . Satisfying this timing is critical because this register is shared between send and receive paths.

The status of an interrupt source is reflected in the **INT_STS** register regardless of whether or not the source is actually enabled as an IRQ interrupt. This register also handles interrupt acknowledgement. Writing a “1” to any bit clears it as long as the interrupt source is no longer active. The interrupt bit will not be cleared if the interrupt condition is still pending regardless of writing 1 to the bit.

The interrupt controller has the ability to generate an IRQ on a Power Management Event when the controller is in the D0, D1 or D2 power states.

5.3 Stopping and Starting the Transmitter

To halt the transmitter, the host can set the **STOP_TX** bit in the **TX_CFG** register. The transmitter will finish sending the current frame (if there is a frame transmission in progress). When the transmitter has pushed the **TX Status** for this frame onto the TX status FIFO, it will clear the **TX_CFG:STOP_TX** and **TX_CFG:TX_ON** bits, and then pulse the **INT_STS:TXSTOP_INT** bit.

Once the transmitter is stopped, the host can clear the TX Status and TX Data FIFOs by setting the **TX_CFG:TXS_DUMP** and **TXD_DUMP** bits. It can also disable the MAC by clearing **MAC_CR:TXEN**. To re-enable the transmitter, the host must set the **TX_CFG:TX_ON** and **MAC_CR:TXEN** bits. When the transmitter is re-enabled, it will begin transmitting any packets that are in the TX DATA FIFO.

5.4 Stopping and Starting the Receiver

To stop the receiver, the host must clear the **MAC_CR:RXEN** bit in the MAC Control Register. When the receiver is halted, the **INT_STS:RXSTOP_INT** bit will be set. Once stopped, the host can optionally clear the **RX Status** and **RX Data FIFO**s. The host can re-enable the receiver by setting the **MAC_CR:RXEN** bit.

5.5 Configuring Address Filtering Options

Table 5.2, "Address Filtering Modes" shows the relations between the control flags which affect the address filter mode.

Table 5.2 Address Filtering Modes

MCPAS	PRMS	INVFILT	HFILT	HPFILT	DESCRIPTION
0	0	0	0	0	MAC Address Perfect filtering only for all addresses.
0	0	0	0	1	MAC Address Perfect filtering for physical address and Hash filtering for multicast addresses
0	0	0	1	1	Hash Filtering for physical and multicast addresses
0	0	1	0	0	Inverse Filtering
X	1	0	X	X	Promiscuous
1	0	0	0	X	Pass all multicast frames. Frames with physical addresses are perfect-filtered
1	0	0	1	1	Pass all multicast frames. Frames with physical addresses are hash-filtered

5.5.1 Configuring Multicast Filtering

Multicast applications act very much like broadcast radio, where all receivers attuned to one station receive the same programming. Multicast packets are a type of broadcast packet, but require filtering to be received by the intended audience.

The hash filtering works by identifying incoming packets with a 48-bit destination multicast MAC addresses, those addresses having a leading bit pattern of 01:00:5e:xx:yy:zz, running only the destination MAC address through a hash (actually it's the CRC32 algorithm, taking the result of the first 6 bytes of the incoming packet and extracting from that the high 6-bits), resulting in a 6-bit (1-of-64) value, which is used to index a 64 x 1-bit array contained across the **MAC HASHH:HASHL** register pair (from bit 0 of **HASHL**:0 on up through bit 31 of **HASHH**:63). If the bit indexed by the hash is a '1', it is considered to be a match, and the entire packet is brought into the device for perfect multicast address filtering in software.

At startup the hash filter is cleared. As the driver accepts requests to join multicast sessions, the driver must add the multicast address to its perfect filter table in software. Then it must calculate the appropriate bit to set in the hash array and update the devices' **HASHH:HASHL** register pair. When the application leaves the multicast session, the driver must update the filter table and the **HASHH:HASHL** register pair appropriately.

5.5.2 Promiscuous Mode

Some applications need to examine every single packet traversing the LAN, whether the packet destination MAC address is broadcast or not, and whether the packet data regards the host in question or not. Communally these programs are referred to as "packet sniffers". An example is the *Ethereal* program, available for a wide range of operating systems (http://www.ethereal.com). At the heart of these programs is the requirement for the LAN device driver to receive every packet on the LAN.

The term for receiving every packet on the LAN is called Promiscuous Mode, and it is supported by all LAN9118 Family members. To enable Promiscuous Mode, set the **MAC_CR:PRMS** bit to a '1' and reset the **MAC_CR:INVFLT** bit to'0'.

5.6 PHY Detection and Initialization

Applications commonly delegate link set-up to the physical media device (PHY). At initialization, the driver can direct the PHY to determine the link parameters by auto-negotiating with its link partner (the link partner is the node on the other side of the Ethernet cable). Using auto-negotiation, the PHY and its link partner determine the optimum speed (10 or 100 Mbps) and duplex setting (full or half) by advertising their capabilities to each other, and then choosing the highest capability that both share in common. Auto-negotiation is documented in the 802.3u Fast Ethernet supplement to the IEEE specification.

Every LAN9118 Family device provides a Media Independent Interface (MII) which connects its internal MAC to an internal PHY. Each PHY must have a unique 5-bit PHY Address; the internal PHY has a PHY Address of 00001b, and this address must be used when addressing the internal PHY. Individual registers within the PHY are indicated through an Index Field. The Table 5.3 and Table 5.4 below show how the **MAC_CSR_CMD** register is used to access the PHY.

Table 5.3 Using the MAC_CSR_CMD Register to Access the MII_ACC Register

MAC_CSR_CMD (ACCESSING MII_ACC REGISTER)			
CSR Busy	R/nW	Reserved (29:8)	CSR Address (7:0)
1	1:0		0x6

Table 5.4 Using the MAC_CSR_CMD Register to Access the MII_DATA Register

MAC_CSR_CMD (ACCESSING MII_DATA REGISTER)			
CSR Busy	R/nW	Reserved (29:8)	CSR Address (7:0)
			0x7

The LAN9117 and LAN9115 provide an external MII interface to support an external PHY, for applications interfacing to other media, such as Fiber, or HPNA. The **MII_ACC: PHY Address** field (bits 15:11) must always correctly reflect the active PHY. If an external PHY is used, it must have a PHY address other than 00000b, 00001b, or 11111b.

Applications which only operate within a fixed environment might consider hardware strapping options instead of auto-negotiating. The SPEED_SEL pin can be used to force the speed setting of the PHY at reset. When the 100 Mbit option is strapped, the PHY will determine whether its link partner is capable of auto-negotiation. If the link partner is capable of auto-negotiation, then the PHY will advertise 100Mbps full and half-duplex capabilities, and use the auto-negotiation process to determine whether the link is full or half-duplex. If the partner is incapable of auto-negotiation, then the PHY will default to a 100Mbps, half-duplex mode of operation. When the 10 Mbit option is strapped, auto-negotiation is disabled, and the PHY defaults to a 10Mbps, half-duplex mode of operation. Software can always override the strap options.

Access to the PHY registers is a bit complex, due to the hierarchical organization of the register set. Recall that the controller contains three sets of registers; the slave set, which are directly accessible to the host processor, the MAC set, which are indirectly accessible to the host processor through a command/data register pair (**MAC_CSR_CMD/ MAC_CSR_DATA**), and the PHY set, which are accessed by the host through a command/data pair of MAC registers (**MII_ACC/ MII_DATA**).

When using the **MAC_CSR_CMD** register to access the MAC Registers (which include the **MII_ACC** and **MII_Data** Registers), the **R/nW** bit selects the data direction for the **MAC_CSR_Data** Register. When the **R/nW** bit is set low, the contents of the **MAC_CSR_Data** Register will be stored into the MAC Register specified by the **CSR Address** field. When the **R/nW bit** is set high, the contents of the MAC Register specified by the **CSR Address** field will be loaded into the **MAC_CSR_Data** Register. The operation starts when the driver sets the **CSR Busy** bit high. The driver should continue polling the **MAC_CSR_Command** Register until the **CSR Busy** bit is low to verify that the operation has completed. A subsequent operation to a MAC Register should not be started until the first operation

has completed, or else spurious operations could result. Setting of the **CSR Address**, **R/nW** and **CSR busy** bits can all be done with a single write command to the **MAC_CSR_CMD** Register.

Table 5.5 Media Independent Interface Access/Command Register

MII_ACC					
Reserved (31:16)	PHY Address (15:11)	MII Register Index (10:6)	Reserved (5:2)	MII W/nR	MII Busy

When using the **MII_ACC** register to access the PHY registers, the **MII W/nR** bit sets the data direction for the **MII_Data** Register. When **MII W/nR** is set low, the contents of the PHY Register addressed by the **PHY_Address** and **MII Register Index** fields will be loaded into the **MII_Data** Register. When the **MII W/nR** bit is set high, the contents of the **MII_Data** Register will be stored into the PHY Register indicated by the **PHY Address** and **MII Register Index** fields. The operation begins when the **MII Busy Bit** is written with a 1 by the host. To verify that the operation has completed, the host should continue polling the **MII Busy Bit** until it is 0. This verification must be completed before any other attempts are made to read or write a PHY register, otherwise invalid operation could result. When initiating a PHY Register access, the entire contents of the MII_ACC Register, including the PHY Address, MII Register Index, MII W/nR bit and MII Busy bit can all be set in a single operation.

When planning access to the hierarchy of register sets, it is a good idea to serialize access to the MAC and PHY registers by utilizing a spin lock (under Linux, or its equivalent). One per device should be sufficient to guarantee that each code sequence in the driver completes without interference from competing driver threads.

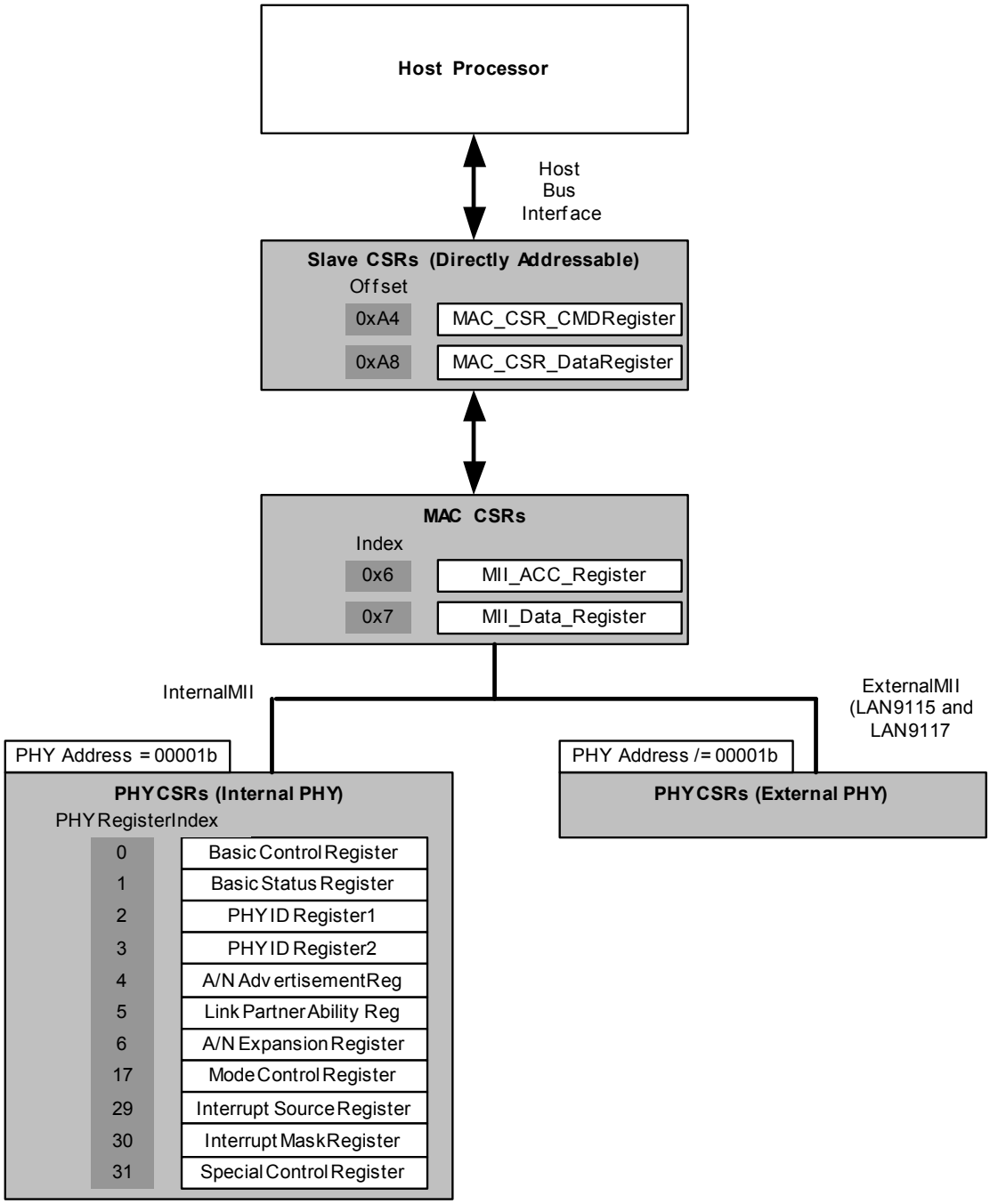

Figure 5.1 PHY Access Command and Data Pathways

The driver can verify the existence of the internal PHY by reading the **PHY Registers 2 and 3 at PHY Address** 1 and ensuring the contents are 0x0007 and 0xc0c1 respectively. The existence of external PHYs is determined by reading the same two PHY Registers at PHY Addresses 0x02 through 0x30. If values of 0xffff and 0xffff are obtained, there is no PHY at the given **PHY Address**.

The driver can reset the internal PHY by writing the **Reset** bit (15) in the **Basic Control Register** (index 0). This bit is self clearing and can be polled (read back) in order to coordinate the PHY reset completion with driver execution. After resetting the PHY, the driver should wait at least 50ms before attempting any further operations on the PHY.

Auto-negotiation begins by enabling all the PHY capabilities in the **Auto-Negotiation Advertisement** (0x1e1), then setting the **Auto-negotiation Enable** and **Auto-negotiate Restart** bits (12 and 9) in the

Basic Control Register (index 0) and completes when the **Auto-negotiate Complete** bit (5) is set in the **Basic Status Register** (index 1). **Auto-negotiate Complete** status should be available within 1.5 seconds. Beyond this time, the driver can then probe for the reason of the failure, such as link down or remote fault, and additionally schedule a thread to revisit the PHY initialization periodically until a proper link has been established.

Once link has been established, the driver can set the **MAC_CR:RDPX** duplex mode to follow the settings of the PHYs **Basic Status Register** (**10Base-T Full Duplex**, **100Base-T Full Duplex**). If auto-negotiation is enabled, then duplex mode can by found by the following sequence:

- AutoNegotiationAdvertisement= ReadPhy(index:4)
- AutoNegotiationLinkPartnerAbility= ReadPhy(index:5)
- AutoNegotiationCompatableSpeeds= AutoNegotiationAdvertisement & AutoNegotiationLinkPartnerAbility
- If (AutoNegotiationCompatableSpeeds & **100Base-T Full Duplex**) <speed is 100 Mbits, full duplex>
- If (AutoNegotiationCompatableSpeeds & **100Base-T**) <speed is 100 Mbits, half duplex>
- If (AutoNegotiationCompatableSpeeds & **10Base-T Full Duplex**)<speed is 10 Mbits, full duplex>
- If (AutoNegotiationCompatableSpeeds & **10Base-T**)<speed is 10 Mbits, half duplex>

Driver writers can derive their own access routines for manipulating the PHY by examining the drivers available from the SMSC website for the LAN9118 Family members. (http://www.smsc.com).

5.7 Switching Between Internal and External PHYs

Drivers for the LAN9117 and LAN9115, which support an external PHY, will need to poll the external MII address space to detect those PHYs. Valid addresses range from 2 to 31, while address 0 is generally considered reserved.

The steps outlined in the flow diagram in Figure 5.2, along with accompanying text, detail the required procedure for switching the MII port, including the MII clocks. These steps must be followed in order to guarantee clean switching of the MII ports.

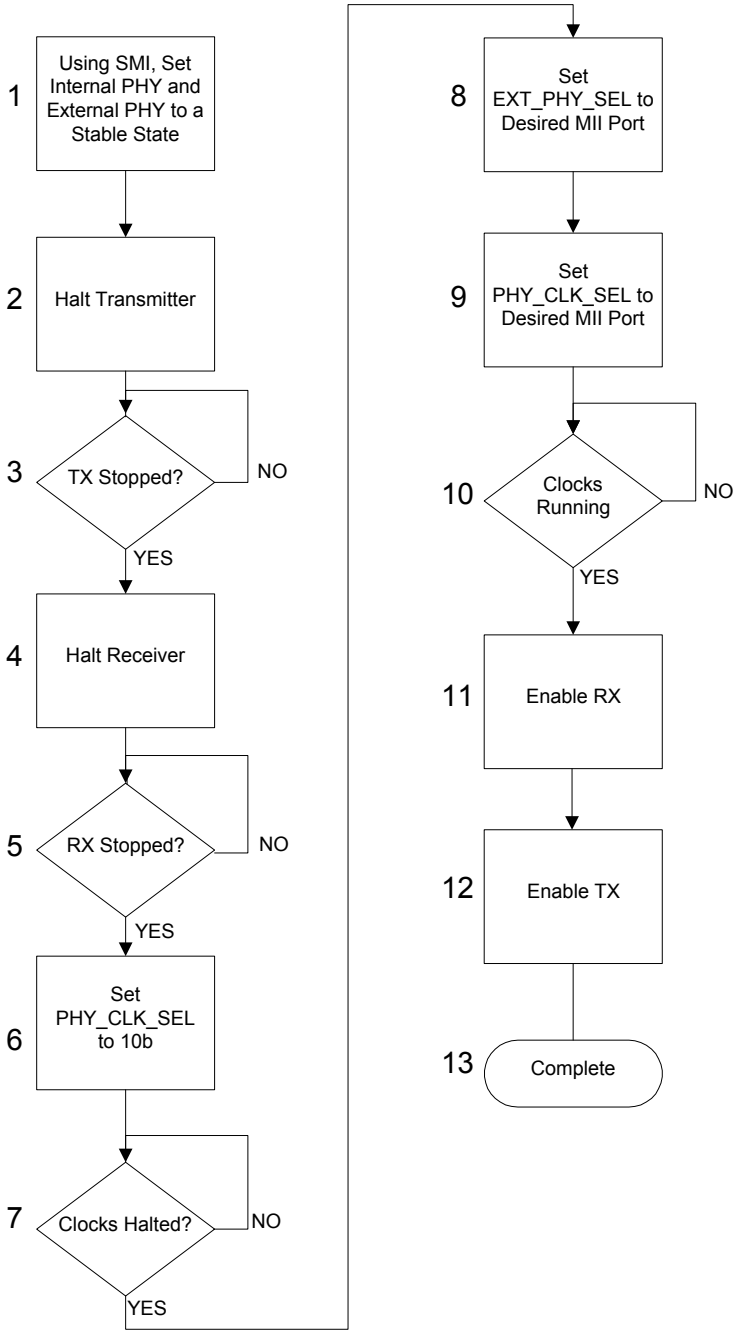

Figure 5.2 The MII Switching Procedure

1. Both the internal PHY and the external PHY must be placed in a stable state. The TX_CLK and RX_CLKs of both devices must be stable and glitch-free before the switch can be made. If either device does not generate a TX_CLK or RX_CLK, this clock must remain off until the switch is complete. In either case the TX_CLK and RX_CLK of the device that will be selected after the switch must be stable and glitch-free.
2. The host must command the transmitter to halt, and the halting of the transmitter must be complete
3. The host must command the receiver to halt, and the halting of the receiver must be complete.

4. The PHY_CLK_SEL field must be set to 10b. This action will disable the MII clocks from the internal and external PHYs to the controller's internal logic.
5. The host must wait a period of time not less than 5 cycles of the slowest operating PHY clock before executing the next step in this procedure. For example, if the internal PHY was operating in 10Mbs mode, and the external PHY was operating at 100Mbs mode, the internal PHY's TX_CLK and RX_CLK period is the longest, and will determine the required wait time. In this case the TX_CLK and RX_CLK period for the internal PHY is 400ns, therefore the host must wait 2us (5*400ns) before proceeding. If the clocks of the device being deselected by the switch are not running, they are not considered in this calculation.
6. Set EXT_PHY_SEL to the desired MII port. This step switches the RXD[3:0], RX_DV, RX_ER, TXD[3:0], TX_EN, CRS and COL signals to the desired port.
7. Set PHY_CLK_SEL to the desired port. This must be the same port that is selected by EXT_PHY_SEL.
8. The host must wait a period of time of not less than 5 cycles of the slowest, newly enabled clock before executing the next step in this procedure.
9. Enable the transmitter.
10. Enable the receiver.
11. The process is complete. The controller is ready to access the new PHY.

The above procedure must be repeated each time the MII port is switched. The procedure is identical whether switching from internal PHY to external MII, or vice-versa.

5.8 Examples of PHY MII Register Reads and Writes

A stated previously, it is better to access the PHY through a pair of routines that access the MAC registers from the **MAC_CSR_** registers, and a pair of routines that access the PHY registers from the **MII_** registers using the **MAC** register routines. The following examples are meant to only to illustrate the concept of accessing the PHY registers.

Example 1: Write the value 0x01e1 to the PHY Auto-Negotiation Advertisement Register (register 4; advertise all capabilities when auto-negotiating)

Step One:

Load the value (0x01e1) destined for PHY Auto-negotiation Advertisement register (4) into the **MAC_CSR_Data** Register. Only the lower 16 bits will be written to the **MII DATA** register, and eventually that will be written to the PHY itself.

MAC_CSR_DATA (TO BE LOADED INTO MII DATA REGISTER	
Reserved (31:16)	PHY Data (15:0)
	0x01E!

Step Two:

Write the command word into the **MAC_CSR_CMD** register. This causes the contents of the **MAC_CSR_Data** register to be written (R/nW == 0) into the **MII Data** register (0x07).

MAC_CSR_CMD			
CSR Busy	R/nW	Reserved (29:8)	CSR Address (7:0)
1	0		0x07

Step Three:

Keep reading the MAC_CSR_CMD Register until the CSR Busy Bit =0.

Load the MII write command word (MII W/nR == 1) that is to be written to the **MII ACC** register into the **MAC_CSR_Data** register. The command word will write the contents of the **MII DATA** register to the PHY Advertisement register (0x4) (note that the lighter-shaded heading is expressed in **MII ACC** register terms).

MAC_CSR_DATA (TO BE LOADED INTO MII ACC REGISTER)					
Reserved (31:16)	PHY Address (15:11)	MII Register (10:6)	Reserved (5:2)	MII W/nR	MII Busy
	0x1	0x4		1	1

Step Four:

Write the command word into the **MAC_CSR_CMD** register. This causes the **MII ACC** register (0x06) to be written (R/nW == 0) with the contents of the **MAC_CSR_Data** register, which in turn causes the MII register write to be executed.

MAC_CSR_CMD (ACCESSING MII_ACC REGISTER)			
CSR Busy	R/nW	Reserved (29:8)	CSR Address (7:0)
1	0		0x06

Steps 5-8:

5. Read the **MAC_CSR_CMD** Register until the **Busy Bit** = 0.

6. Load the **MAC_CSR_CMD** Register as follows: **CSR Busy** = 1, **R/nW** = 1 (Read), **CSR Address** = 0x06 (**MII_ACC** Register)

7. Read the **MAC_CSR_CMD** Register until the **Busy Bit** = 0

8. Read the **MAC_CSR_DATA** Register. Repeat steps 6-8 until the **MII Busy Bit** (Bit 0) = 0.

Example 2: Read the PHY Status register (register 1)

Step One:

Load the **MAC_CSR_Data** register with the **MII ACC** register command word to read (MII W/nR == 1) PHY register 1 (MII Register == 1). Note that the lighter-shaded heading is expressed in **MII ACC** terms.

MAC_CSR_DATA					
Reserved (31:16)	PHY Address (15:11)	MII Register (10:6)	Reserved (5:2)	MII W/nR	MII Busy
	0x1	0x1		0	1

Step Two:

Write the command word into the **MAC_CSR_CMD** register. This causes the contents of the **MAC_CSR_Data** register to be written (R/nW == 0) to the **MII ACC** register (0x06), which executes the PHY read. The resulting read from the PHY is then loaded to the **MII Data** register

MAC_CSR_CMD (ACCESSING MII_ACC REGISTER)			
CSR Busy	R/nW	Reserved (29:8)	CSR Address (7:0)
1	0		0x06

Steps Three - Seven:

3. Read the **MAC_CSR_CMD** Register until the **CSR Busy** Bit = 0.

4. Write the **MAC_CSR_CMD** as follows: **Busy Bit** = 1, **R/nW** = 1, **CSR Address** = 0x06.

5. Read the **MAC_CSR_CMD** Register until the **CSR Busy Bit** = 0

6. Read the **MAC_CSR_Data** Register. Repeat Steps 3-6 until the MII Busy Bit = 0 The contents of the PHY Status Register (Register 4) have now been loaded into the MII_Data Register.

Step Seven

Write the command word into the **MAC_CSR_CMD** register. This command causes the **MAC_CSR_Data** register to be loaded (R/nW == 1) with the contents of the **MII Data** register (0x07).

MAC_CSR_CMD			
CSR Busy	R/nW	Reserved (29:8)	CSR Address (7:0)
1	1		0x07

Steps Eight-Nine:

8. Read the MAC_CSR_CMD Register until the CSR Busy Bit = 0

9. Read the **MAC_CSR_Data** register, which now contains the contents of the **MII Data** register in the lower 16-bits.

MAC_CSR_DATA	
Reserved (31:16)	PHY Data (15:11)

6 Transmit Packet Processing

Transmitting packets is a straightforward process compared with receiving packets; in that the host can efficiently synchronize the entire transmit process with the application. In practice though, interrupts are still needed for synchronizing to the time-dependant embedded applications.

Memory Alignment Technology (MAT) in the LAN9118 family allows the driver to optimize the data flow between packet data memory and the device FIFO by allowing the driver to always perform aligned, efficient data transfers, regardless of the packet data organization in memory. This is achieved by providing a descriptor along with the data, one for each transfer, called a Command Word that indicates the true alignment of the packet, along with an indicator of the underlying hardware nature of the transfer.

6.1 Transmit Data Transfer

At the heart of the transmit packet process, two DWORD values called **TX_CMD_A** and **TX_CMD_B** command words are written into the device transmit data register (**TX_DATA_FIFO**), followed by the packet buffer data. Host devices using Programmed IO (PIO) for data transfers work by copying packet data in 4-byte increments aligned to 4-byte address boundaries. DMA-controllers generally transfer data in 4-, 16- and 32-byte increments that are correspondingly aligned in memory . On the other hand, the OS data buffers may begin on arbitrary byte boundaries, creating misaligned transfers with respect to the physical addressing and caching of the CPU or DMA Controller. Memory Alignment Technology (MAT) uses a description of the misalignment to allow the hardware to compensate for the misaligned data buffers.

Take a moment to notice that the transmit data FIFO register is aliased to 8 contiguous DWORD locations. The purpose of this aliasing is to accommodate DMA controllers which transfer data in 16- and 32-byte bursts. This is not meant to imply that misaligned data transfers need to write to misaligned addresses within the device; that alignment is handled in the software construction of the data transfer base address. Rather that the writer should be mindful that some CPUs and DMA engines will generate extra, sequential addressing cycles to memory; use of the term “extra” is only from the driver’s perspective. Phantom addressing cycles might be a more appropriate term, such as when the DMA engine transfers a 32-byte burst of data to the device, creating 8 x DWORD accesses, each on a unique, ascending DWORD boundary.

6.2 Command Word Construction

The **TX_CMD_A** command word contains six fields of concern:

1. **Buffer End Alignment** is a 2-bit field which controls device behavior with a DMA controller. As mentioned above, PIO transfers always use 4-byte alignment, which results in the packet data buffer transfer occurring in a multiple of DWORD transfers; the last DWORD transfer may contain pad bytes, depending upon the size of the transfer, modulo 4.

   In addition to PIO transfer, LAN9118 Family devices are designed to take advantage of the more efficient data transfer modes afforded by the use of a DMA controller. When using DMA with the device, burst transfers where the read/write access to the device is burst across 4 or 8 contiguous DWORD addresses, the device’s aliasing of the **TX Data FIFO** register presents a simple memory interface to the DMA controller. The setting of End Alignment causes the device to insert pad bytes into the final transfer, filling the remainder of the 16- or 32-byte transfer, but without causing these bytes to end up in the transmit data stream.

   Data Start Offset is a 5-bit field which indicates to the device the number of bytes offset from the aligned, base-address where the data actually begins. **Table 4-2** shows how to construct the **Buffer End Alignment** and **Data Start Offset** fields. In a 4-byte example, the data transfer begins at physical address YYYY, with its lowest 2 address bits cleared (xx == 00b).

   First Segment is a 1-bit indication that the data transfer marks the beginning of a single packet.

Last Segment is also a 1-bit field, marking the data transfer as the end of a single packet.

To better understand the use of these last two fields, let's discuss two popular operating systems that utilize different packet allocation schemes. Under Linux, system kernel buffers (*skb*'s) are used to hold packet data and almost always contain enough space for the largest possible packet size (1514 bytes) as an optimization of time-of-allocation. For sending a packet in a single data buffer transfer operation, these two fields (**First Segment**, **Last Segment**) would both be set.

FreeBSD on the other hand uses an *mbuf* (64-byte memory management buffers) packet allocation scheme to economize memory allocation space by chaining together only as much *mbuf* memory as needed to contain a given packet. Each *mbuf* typically holds the header from a single layer (MAC, IP, UDP, or RTP). Under this scheme each *mbuf* could be dealt with as an individual data transfer, and the head link in the chain would have the **First Segment** flag set in its command word, while the tail link would have **Last Segment** set. Intervening *mbufs* will have neither flag set (see **TX Buffer Fragmentation Rules** in the datasheet for more detail).

Buffer Size is an 11-bit field specifying the number of data bytes in the current data transfer not including the two command words.

Interrupt on Completion causes **TX_DONE** to be asserted after the current data transfer has completed. This is useful and important for chained DMA transfer operations.

The **TX_CMD_B** command word contains 2 major and 2 minor fields of concern:

1. **Packet Tag** is a 16-bit field which contains a unique ID which can be read back from the **TX Status** word after the packet has been sent. This field is examined by the device only to the extent that in a chained packet transfer where all transfers relate to a single packet, all tags must be identical, but beyond that, the tag may be used for a variety of purposes by the driver writer, such as tracking the chains, or keeping byte count statistics.
2. **Packet Length** contains the overall number of bytes in the current packet. Note the distinction to the **Buffer Size** field in TX_CMD_A. All links in an *mbuf* scheme will contain an identical value in this field.

- **Add CRC Disable** disables the automatic appending of CRC to the packet.
- **Disable Ethernet Frame Padding** prevents the automatic padding of the packet when the overall packet length is less than 64 bytes (IEEE 802.3). A side affect is that CRC is always added, despite the setting of **ADD CRC Disable**.

6.3 Per Packet Flow Control

To regulate data transfers on a per-packet basis, a driver needs to track the space remaining in the TX_FIFO for subsequent transmit packets (flow design works better if the packets which the device is currently transmitting do not have to be stalled). This can be obtained by reading the **TX Data FIFO Free Space** field (**TDFREE**) of the **Transmit FIFO Information** register (**TX_FIFO_INF**). Note that **TDFREE** is read in bytes and represents the free space in the **TX Data FIFO**. The device offers the driver writer a variety of techniques to synchronize the application with the device having enough free space to accommodate a packet transfer. One method of handling the situation where there is not enough free space to accommodate the current packet transfer is for the driver to enable the device to interrupt when the free space exceeds a threshold level specified by the **FIFO_INT:TX Data Available Level** field, and enabling the **INT_EN:TDFA_INT_EN** bit. Alternately a scheme could be constructed using the **INT_EN:TSFL_INT_EN** interrupt which signals a level change in the **TXUSED** count. In this scheme the driver would track the number of TX_FIFO bytes freed up simply by summing the bytes used in packets completed, and deciding the appropriate moment to re-enable the transmit packet flow. Use of the FIFO interrupt schemes is encouraged, as it helps eliminate register sharing between transmit and receive threads.

6.4 Packet Transfer Completion: Management Statistics

Once a data packet has been transferred into the **TX_Data_FIFO**, most protocol stacks have no need for the sending thread to acknowledge the transfer result; the upper protocol layers are expected to deal with failures, such as a TCP layer re-transmit. There is a need in a managed host however, to maintain statistics per device such as the number of packets or bytes sent, the number of packet that have been deferred, collisions, under runs, etc. After a packet has been processed by the device, a **TX Status** word is appended to the end of the device's **TX_Status_FIFO** for checking individual packet transfer status whether the packet has been transmitted or dropped; the driver can update the transmit statistics periodically from a separate thread or call to examine these values. The **Packet Tag** field is copied from the **TX_CMD_B** word used in the packet transfer, while **Error Status** is a 1-bit summation of the remaining **TX Status** error flags; 0 means no errors, i.e., that the packet has been sent.

Table 6.1 ransmit Status Word

TX STATUS																
Packet Tag (31:16)	Error Status	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1234																

The **TX_FIFO_INF:TSUSED** field gives a count of many **TX Status** words are pending to be read from the **TX_STATUS_FIFO** register. A status word is added to the Status FIFO whenever a packet completes processing, whether or not the packet was transmitted successfully. The count field does not account for packets that are currently in the Transmit Data FIFO and have not yet been transmitted. Use of the count field and its associated interrupt allows a driver some flexibility towards management style. If no management is required, the driver can choose to ignore status word processing completely and save code space and processing time. The strictest style of management can be achieved by programming the FIFO to stop sending packets when the **TX_Status_FIFO** is full, and examining every status word until **TSUSED** is again 0. A looser style of management could also be achieved by allowing the **TSUSED** counter to overrun, and thus simply make a "best effort" at managing the device.

6.5 Transmit Packet Examples

Let's take a look at Table 6.2, "Transmit Packet Example", which shows a 78-byte (0x4e) transmit packet used in the example, which begins at location 0xb4002013 in physical memory. In this case, it happens to be an IP packet to be sent from our MAC source address of 00:80:0f:71:55:71 to a destination MAC address of 00:0e:83:a0:de:ff.

Table 6.2 Transmit Packet Example

	PHYSICAL MEMORY															
0XB4002000	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
0x00																
0x10	XX	XX	XX	00	0e	83	a0	de	ff	00	80	0f	71	55	71	08
0x20	00	45	00	00	28	11	56	40	00	80	06	f0	6a	aa	81	50
0x30	76	aa	81	53	96	04	e7	01	bd	a9	4a	38	0d	0a	d6	7c
0x40	6c	50	10	Ff	ff	f9	29	00	00	66	6f	72	53	4d	53	43
0x50	4d	2e	20	44	62	76	69	64	20	47	65	4c	62	6d	61	6e
0x60	21	YY	YY	YY												
0x70																
0x80																

In a PIO type transfer, 4-byte DWORD values are copied from memory to the device **TX_DATA_FIFO**, so we begin building the **TX_CMD_A** command word by setting the **Buffer End Alignment** field to 0x0 to indicate 4-byte alignment:

Table 6.3 Buffer End Alignment

ALIGNMENT REQUIREMENT	VALUE (25:24)
4-word alignment	0x0
16-word alignment	0x1
32-word alignment	0x2
(undefined, illegal)	0x3

Then, use the 2 low-bits from of the physical address (0x3) to set the 5-bit **Data Start Offset** field, since we are only transferring 4 bytes at a time (PIO). Also we want to transfer the entire packet in a single buffer transfer action, so **First Segment** and **Last Segment** flags are both set.

In the **TX_CMD_B** command word, we use a unique value of 0x1234 for the **Packet Tag** field, and an overall **Packet Length** the same as the **Buffer Size** as the entire packet is transfer in a single action. Note that the **Buffer Size** refers to the individual transfer payload, not counting the offset and padding alignment, but in this simple case, length and size are the same.

TX_CMD_A															
31	30	29	28	27	26	Buffer End Alignment	23	22	21	Data Start Offset	15	14	First Segment	Last Segment	11	Buffer Size
						0x0				0x3			1	1		0x4e

Table 6.4 Transmit Command Words

TX_CMD_B						
Packet Tag	15	14	13	12	11	Packet Length
0x1234						0x4e

Keep in mind that this example is running in a Little Endian environment. To begin the transfer sequence, write the two command words as DWORDs, followed by the packet data. Since the transfer is aligned on 4-byte boundaries, the data transfer start address must begin from 0xb4002010 and continue up to and including 0xb4002060. Table 6.5, "Single Buffer Single Packet Data Transfer Sequence" shows the sequence of data transferred from Table 6.2: Transmit Packet Example's memory into the **TX_DATA_FIFO**. To read the Table 6.5, start from the upper left portion, and following the Host Source Address (light grey) headings down, then zigzag back up and down through the middle, and then the right-hand columns, so that the transfer appears thus:

Table 6.5 Single Buffer Single Packet Data Transfer Sequence

HOST SOURCE ADDRESS	TX_DATA_FIFO DATA TRANSFER 1	HOST SOURCE ADDRESS	TX_DATA_FIFO TRANSFER CONT.	HOST SOURCE ADDRESS	TX_DATA_FIFO TRANSFER CONT.
TX_CMD_A	0x0003304e	0xb4002028	0xf0068000	0xb4002048	0x726f6600
TX_CMD_B	0x1234004e	0xb400202c	0x5081aa6a	0xb400204c	0x43534d53
0xb4002010	0x00XXXXXX	0xb4002030	0x5381aa76	0xb4002050	0x44202e4d
0xb4002014	0xdea0830e	0xb4002034	0x01e70496	0xb4002054	0x64697662
0xb4002018	0x0f8000ff	0xb4002038	0x384aa9bd	0xb4002058	0x4c654720
0xb400201c	0x08715571	0xb400203c	0x7cd60a0d	0xb400205c	0x6e616d62
0xb4002020	0x00004500	0xb4002040	0xff10506c	0xb4002060	0xYYYYYY21
0xb4002024	0x40561128	0xb4002044	0x0029f9ff		

The second example will demonstrate how to chain transfers together into a single packet. We will re-use the sample packet shown in Table 6.2, and instead of a single transfer, we will make 3 separate transfers; 29 bytes to start, then 32 bytes, then the remaining 17 bytes. Visually the chained transfer appears like this:

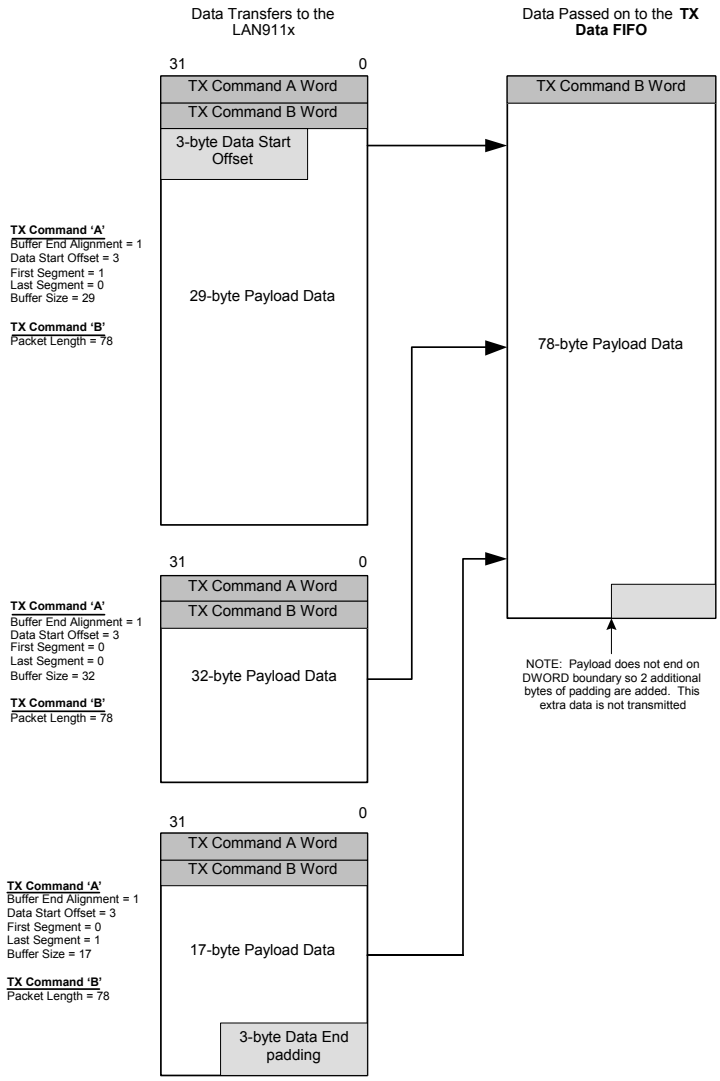

Figure 6.1 Multiple (3) Buffer Data Single Packet Transfer Sequence

6.5.1 Transmit Command Words for Figure 6.1, "Multiple (3) Buffer Data Single Packet Transfer Sequence"

Table 6.6 TX_CMD_A for Segment 1

31	30	29	28	27	26	BUFFER END ALIGNMENT	23	22	21	DATA START OFFSET	15	14	FIRST SEGMENT	LAST SEGMENT	11	BUFFER SIZE
						0x0				0x3			1	0		0x1d

Table 6.7 TX_CMD_B for Segment 1

PACKET TAG	15	14	13	12	11	PACKET LENGTH
0x5678						0x4e

Table 6.8 TX_CMD_A for Segment 2

31	30	29	28	27	26	BUFFER END ALIGNMENT	23	22	21	DATA START OFFSET	15	14	FIRST SEGMENT	LAST SEGMENT	11	BUFFER SIZE
						0x0				0x0			0	0		0x20

Table 6.9 TX_CMD_B for Segment 2

PACKET TAG	15	14	13	12	11	PACKET LENGTH
0x5678						0x4e

Table 6.10 TX_CMD_A for Segment 3

31	30	29	28	27	26	BUFFER END ALIGNMENT	23	22	21	DATA START OFFSET	15	14	FIRST SEGMENT	LAST SEGMENT	11	BUFFER SIZE
						0x0				0x0			0	1		0x11

Table 6.11 TX_CMD_B for Segment 3

PACKET TAG	15	14	13	12	11	PACKET LENGTH
0x5678						0x4e

Note: The use of the **First Segment** and **Last Segment** flags in the example detailing the head, middle, and tail of the chained transfer. Also see how the **Buffer Size** is unique for each transfer, while the **Packet Length** reflects the total packet size. Finally, the **Packet Tag** must be identical for correct operation. The new sequence of 3 data transfers looks like this:

Table 6.12 Multiple (3) Packet Data Transfer Sequence

HOST SOURCE ADDRESS	TX_DATA_FIFO DATA TRANSFER 1	HOST SOURCE ADDRESS	TX_DATA_FIFO DATA TRANSFER 2	HOST SOURCE ADDRESS	TX_DATA_FIFO DATA TRANSFER 3
TX_CMD_A-1	0x0003201d	TX_CMD_A-2	0x00000020	TX_CMD_A-3	0x00001011
TX_CMD_B-1	0x5678004e	TX_CMD_B-2	0x5678004e	TX_CMD_B-3	0x5678004e
0xb4002010	0x00XXXXXX	0xb4002030	0x5381aa76	0xb4002050	0x44202e4d
0xb4002014	0xdea0830e	0xb4002034	0x01e70496	0xb4002054	0x64697662
0xb4002018	0x0f8000ff	0xb4002038	0x384aa9bd	0xb4002058	0x4c654720
0xb400201c	0x08715571	0xb400203c	0x7cd60a0d	0xb400205c	0x6e616d62
0xb4002020	0x00004500	0xb4002040	0xff10506c	0xb4002060	0xYYYYYY21
0xb4002024	0x40561128	0xb4002044	0x0029f9ff		
0xb4002028	0xf0068000	0xb4002048	0x726f6600		
0xb400202c	0x5081aa6a	0xb400204c	0x43534d53		

6.6 The Overall Packet Transmit Process

Now that we've established a background for transmitting packets with the device, let's look at the diagram in Figure 6.2, "Packet Transmission" depicting a packet transmit scheme. This example is based upon the SMSC simple Linux driver, which provides driver calls to block and un-block the transmit queue which calls into the driver:

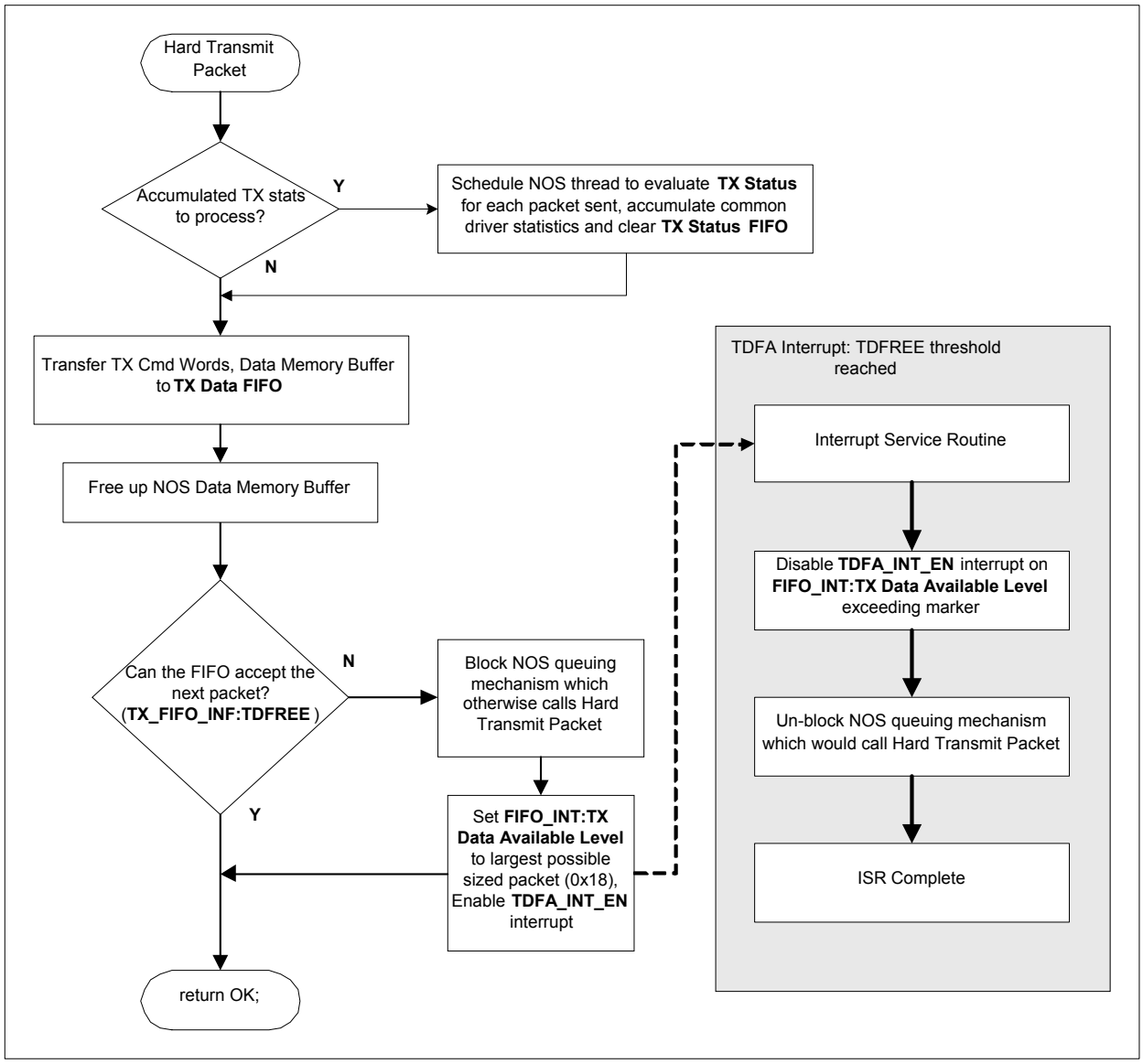

Figure 6.2 Packet Transmission

Once the device is initialized, the application can continue sending packets until the first indication appears that the device can no longer accept packets (since we cannot know the size of the next transmit packet a priori, we want to ensure that we can send the largest packet possible, 1518 bytes in size). At that point we tell the network operating system (NOS) to stop sending the driver any more transmit packets. When the device has freed up enough buffer space, due to the transmitting of packets internally queued, it interrupts the application, which disables the interrupt and calls into the NOS telling it that we can now resume accepting packets for transmit.

For simplicity and completeness, this example shows the TX status results processing handled as a thread scheduled directly at the entry of the packet transmit function. In truth this thread of execution could be scheduled from its own **TX Status FIFO Level Interrupt (TSFL),** which signals that a threshold number of packets have completed the transmit process (successful or not).

7 Receive Packet Processing

A receive process must signal the host that there are incoming packets to be read in, it must identify the validity and length of each individual packet, copy the packet data from device to memory, hand off the packet data memory to the protocol stack, maintain count statistics, and continue the application when there are no more incoming packets. This is much like transmit processing in reverse, with the difference mainly being one of synchronicity. The low-level transmit routine sends a packet whenever the application offers it one; the transmitter is ready to accept the packet most of the time. The receive processing on the other hand, does not know a priori when a packet is being received. Hence there is a need to drive the process asynchronously via the device **RX Status FIFO Level** (**RSFL**) interrupt, which signals the NOS that there are incoming packets to be serviced. Polling could be employed by a thread to examine the same device status as that causing the interrupt, but it is better practice and more efficient if the driver relies upon one of the available interrupts, either by detecting the change in time, such as the **GP Timer**, or status change. Sometimes the underlying operating system can avail the driver a useful polling API, such as NAPI under Linux, but in any case, the driver author can choose from a variety of techniques to synchronize the ingress of packet data to the application.

The first information the driver needs for each packet arriving at the device is its associated receive status word (**RX Status**). We are primarily concerned with the length of a packet (**Packet Length**) and any indications of errors. Like the **TX Status: Error Status**, the **RX Status:Error Status** bit is a quick summation of typical receive errors, such as runts, frames too long, late collisions, and CRC errors. Using the remaining status information allows a driver other options, including a quick discrimination between Ethernet, 802.3, broadcast and multicast frames. While discussing the receive status word, keep in mind that when the **MAC_CR:PADSTR** bit is set, the **Packet Length** reflects the data actually sent across the wire, and CRC is stripped off of the packet (0x5ea). Otherwise packets less than 64-bytes are padded out to 64-bytes and CRC is always appended.

Table 7.1 Receive Status Word

RX STATUS																	
31	30	Packet Length (29:16)	Error Status	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
		0x59	0															

To detect the presence of incoming packets, the receiver maintains a queue of status words, which can be individually read from the **RX Status FIFO** port, each corresponding to a packet positioned in the data queue. Each read pops a status word off the front of the queue. Some driver environments may require the driver to examine the status before popping it off of the queue, and for this a **RX Status FIFO PEEK** port is provided. At any given moment, the **RX_FIFO_INF:RXSUSED** register field gives a current, reliable count of status words in the FIFO, and hence how many packets, good or bad, are queued up for processing. This field has associated with it an **RSFL_INT_EN** interrupt bit and threshold count **FIFO_INT:RX Status Level** register field, which can be programmed to signal the host whenever the current count exceeds the threshold value.

The **RSFL** interrupt handler depicted in Figure 7.1, "Packet Reception", should acknowledge and disable the interrupt, and then schedule a thread to run outside of interrupt state, to minimize the amount of time interrupts are blocked in the application. When the thread has read in all of the pending receive packets, the interrupt can again be enabled.

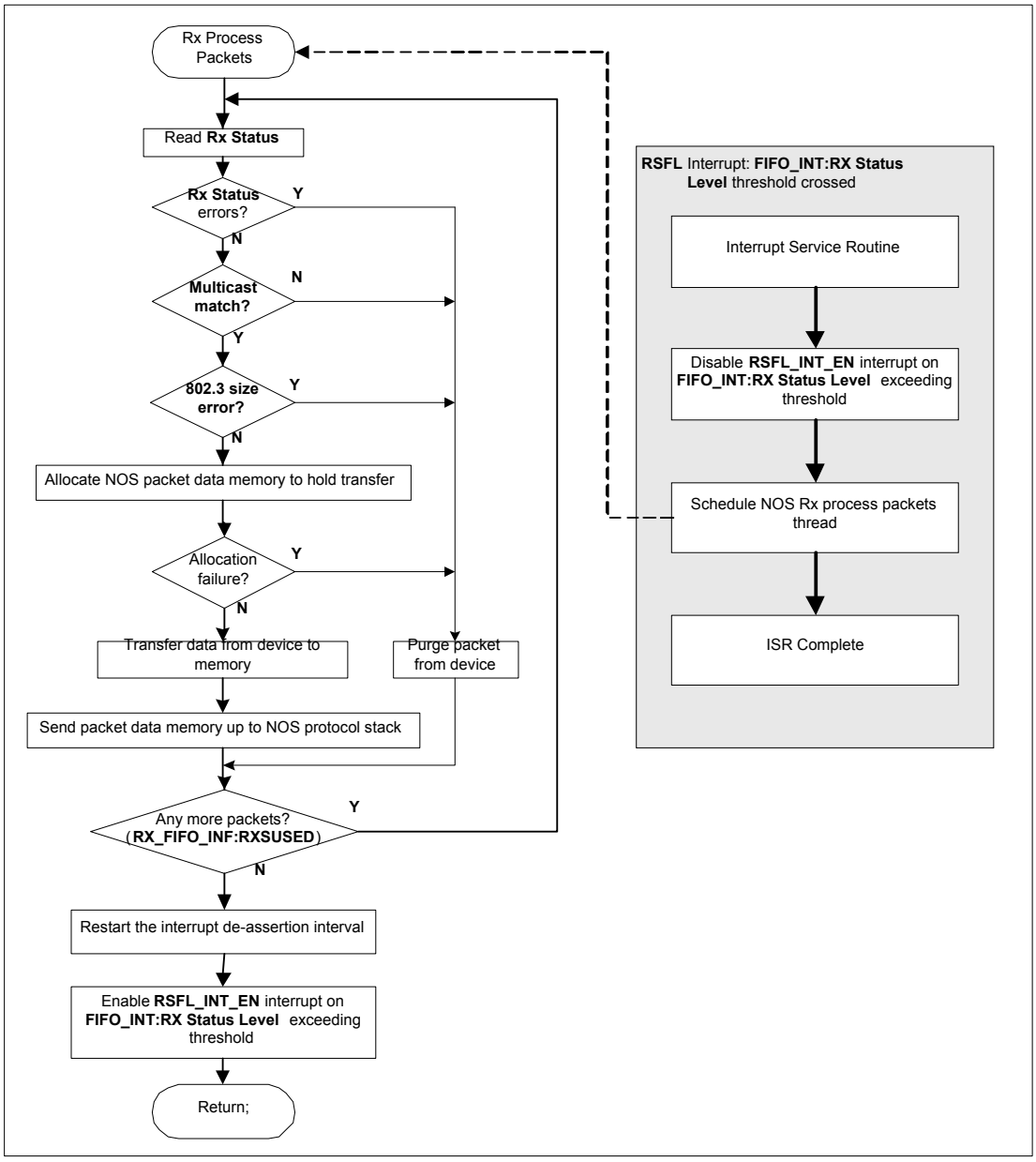

Figure 7.1 Packet Reception

In the Figure 7.1 above we only begin reading packets whenever the **RX Status FIFO** contains any entries (see **RX_FIFO_INF:RSXUSED**). Then, as long as we have more packets remaining in the device, we check the individual **RX Status** per packet, and try to allocate a data packet in memory from the network operating system. If everything is OK, then we can transfer the packet from our device into packet data memory, and hand the packet up to the protocol stack for application processing. Should any errors in status or an inability to obtain data memory be encountered in the process, the driver will need to purge the device of the bad packet.

If Multicast reception is required in the application, then the driver may be the point at which to validate the destination multicast address against a current list of multicast addresses acceptable to the application. Although the hardware supports hashed multicast address matching, perfect multicast filtering match must be performed, whether it's in the in the driver, or in the upper layers of the protocol stack. Should 802.3 style frames be an issue for the application, the driver can also evaluate the length/type field and validate it against the device's notion of the true, physical packet length.

7.1 Receive Data Transfer

Given that the device makes all data transfers in DWORD pieces, the driver can easily align the receiving data memory address to a DWORD boundary. Recall too, that the data transfer length must also be rounded up when considering overall data buffer length (i.e., a 109-byte packet should be treated as a 4*28 or 112 byte transfer).

The **RX Data Offset** field gives the driver control over packet alignment

Table 7.2 Receive Data Buffer Example

	PHYSICAL MEMORY															
0XB400200 0	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
0x00																
0x10			00	0e	83	a0	de	ff	00	80	0f	71	55	71	08	00
0x20	45	00	00	28	11	56	40	00	80	06	f0	6a	aa	81	50	76
0x30	aa	81	53	96	04	e7	01	bd	a9	4a	38	0d	0a	d6	7c	6c
0x40	50	10	ff	ff	f9	29	00	00	66	6f	72	53	4d	53	43	4d
0x50	2e	20	44	62	76	69	64	20	47	65	4c	62	6d	61	6e	20
0x60	77	72	6f	74	65	20	74	68	69	73	2e					
0x70																
0x80																

Commonly, TCP/IP based protocol stacks optimize packet data access by aligning the start of the IP header to a DWORD boundary. Using the example **RX Status** of an 89-byte packet shown in Table 7.2 just above is a packet buffer starting at 0xb4002000, and it shows where we want the data to wind up after the transfer is complete, The data buffer is aligned on a 32-byte boundary, and contains an IP version 4 style packet, with a header length of 5 DWORDS (0x45), After the transfer, the beginning of the IP header portion, the byte containing the value 0x45 is aligned at physical address 0xb4002020. As the MAC header which precedes it in memory is 14 bytes in length (6-byte destination address, 6-byte source address, 2-byte packet type/length), adding an 18-byte offset to the start of where the driver stores this packet, will align the IP header on the next 32-byte boundary (14+18 == 32). For this example we will be working with a DMA controller capable of 32-byte cache line burst transfers, hence the need for **RX End Alignment**. It is necessary for the driver to inform the device of the DMA controllers burst ability because the burst addressing will cause additional reads to be generated to the device even though no data may actually be transferred at the end of the packet transfer. Hence the need for padding the packet memory buffer to the alignment indicated by this field

:

Table 7.3 RX End Alignment

ALIGNMENT REQUIREMENT	VALUE (31:30)
4-word alignment	0x0
16-word alignment	0x1
32-word alignment	0x2
(undefined, illegal)	0x3

Enabling a receive transfer requires setting up the **RX_CFG** register, shown in Table 7.4, "Receiver Configuration Word" below. For this example, the **RX_CFG: RX Data Offset** is given a value of 18 (0x12), which gives the starting alignment of the MAC header which is needed to place the IP header on the DWORD address of 0xb40020020. For PIO transfers, DWORD end alignment is sufficient, so this field would be set with a value of '00' to indicate such. But since we are doing this transfer using a DMA controller capable of cache line bursting, we can set the **RX End Alignment** to reflect the controller's burst transfer ability, in this case 32-byte ('10' == 2) end alignment.

Table 7.4 Receiver Configuration Word

RX_CFG															
RX End Alignment	29	28	RX DMA Count	15	14	13	RX Data Offset	7	6	5	4	3	2	1	0
0x2			0x20				0x12								

When the driver is written to work along with a DMA controller, it becomes necessary to coordinate the completion of a transfer. Usually this coordination is based upon the DMA controller interrupting on terminal count (all bytes transferred), but there are situations where we want the driver to continue receive processing in parallel to the DMA operations, particularly if the DMA controller is descriptor-based. The device will assert the **RXD_INT** interrupt to the host on terminal count by programming the **RX_CFG:RX DMA Count** field with the transfer size. As this field is 12-bits wide and the count is expressed in DWORDS, the driver can potentially start a series of transfers and continue only after they have all completed. For this example we need to calculate the **RX DMA Count** by adding the **RX Data Offset** preceding the start of the packet **(**0x12) to the **Packet Length** of 0x59 and finish by adding the number of padding bytes implied by the **RX End Alignment** (from offset 0x6b through 0x7f == 0x15) for a sum total of 18 + 89 + 21 == 128 bytes, or 32 DWORDS (0x20).

Having written the **RX_CFG** value we can enable the DMA controller to transfer 128 bytes from the **RX Data FIFO** to the packet memory buffer beginning at address 0xb4002000. The device will handle the correct data alignment for the transfer to result in the example shown in Table 7.2. Since we assigned a non-zero value to the **RX DMA Count**, we are not counting on the controller to interrupt us on its terminal count. The device will signal transfer complete by raising the **RXD_INT** interrupt.

7.2 Purging Receive Packets

There are valid reasons for throwing away data in the normal course of driver operations, such as malformed data or a lack of data memory buffers in which to hold and process the data. Another reasons is to do perfect multicast filtering, which is performed in software. Since only the destination address is required for evaluation, the driver could read in 2 DWORDS, and decide whether or not to bring the rest of the packet in. Another example is IEEE 802.3 frame size errors, where the MAC header length/type field does not correspond with the value in the **RxStatus:Packet Length** field

A packet is purged by setting the **RX_DP_CTRL:RX_FFWD** bit, and then polling it (read back and test for bit reset to 0) until it has reset. If the packet is less than 4 DWORDS in length however, then the

RX_DATA_FIFO must be read an appropriate number of DWORDS to purge the miniscule packet. This should only happen if the **MAC_CR** is configured to **PASSBAD** packets (== '1').

7.3 Flow Control Function

The flow control function monitors space available in the **RX Data FIFO**. When the threshold set by the in **AFC_CFG:AFC_HI** field is reached, the Flow Controller will instruct the MAC to send a pause frame (in full-duplex mode), or to jam incoming frames (in half-duplex mode). Logic residing in the MAC is then responsible for inserting the PAUSE frame or generating backpressure at the appropriate time.

7.3.1 Backpressure

When operating in half-duplex mode with the transmitter and flow control enabled, the controller will automatically assert backpressure based on the setting of **AFC_CFG** register. When the **RX Data FIFO** reaches the level set by **AFC_HI** the Backpressure Duration Timer will start. The controller will assert backpressure for any received frames as defined by the **FCANY**, **FCADD**, **FCMULT** and **FCBRD** control bits. This continues until the Backpressure Duration Timer reaches the time specified by the **BACK_DUR** field. After the **BACK_DUR** time period has elapsed the receiver will accept one frame. If, after receiving one RX frame, the **RX Data FIFO** is still above the threshold set by **AFC_LO**, the controller will again start the Backpressure duration timer and will assert backpressure for subsequent frames repeating the process described here until the **RX Data FIFO** level drops below the **AFC_LO** setting. If the **RX Data FIFO** drops below **AFC_LO** before the Backpressure Duration Timer has expired, the timer will immediately reset and backpressure will not be asserted until the **RX Data FIFO** level exceeds **AFC_HI**.

If the **AFC_LO** value is set to all ones (0xff) and the **AFC_HI** value is set to all zeros (0x00), the flow controller will assert backpressure for received frames as if **AFC_HI** threshold is always exceeded. The driver can use this mechanism to direct flow control by enabling and disabling the **FCANY**, **FCADD**, **FCMULT** and **FCBRD** bits.The **BACK_DUR** field can be used to avoid excessive collisions, and the **FCANY** bit can be used to eliminate late collisions, unless the LAN topography is known before hand.

7.3.2 Pause Frames

When operating in full-duplex mode with the transmitter and flow control enabled, the controller will automatically transmit pause frames based on the settings of **AFC_CFG** and **FLOW** registers. When the **RX Data FIFO** reaches the level set by **AFC_HI** the controller will transmit a pause frame. The pause time field that is transmitted is set in the Pause Time (**FCPT**) field of the **FLOW** register. When the **RX Data FIFO** drops below **AFC_LO** the controller will automatically transmit a pause frame with a pause time of zero. The controller will only send another pause frame when the **RX Data FIFO** level falls below **AFC_LO** and then exceeds **AFC_HI** again.

8 Instrumentation and Debug

This chapter is based upon the experience gained from the development of the simple Linux Driver for the LAN9118 family. It assumes that the driver will have a rich feature set in its run-time environment to draw upon for debugging. While not every operating system offers the functionality of a Linux, the concepts remain useful and are offered to the driver developer as tips.

When something goes wrong in the driver, it is very helpful to have facilities for observing its internal workings. This chapter will describe the various methods available to the driver writer.

8.1 Debug Prints

One of the simplest methods of debugging is Debug Prints. These are similar to printf() statements that might be used by any software. They can be used to reveal the contents of a variable or a register at some special location during code execution. They are useful when the driver is behaving unexpectedly. Debug prints should not be left in performance-sensitive paths of the driver, because debug prints are down execution tremendously. Debug prints can cause “Heisenberg-bugs”, since their very presence alters timing. They may mask or even induce problems.

The simple driver uses the following macros for debug prints. See simp911x.c for details.

8.1.1 SMSC_TRACE(message, parameters)

SMSC_TRACE points work just like a printf() statement, in fact they call the Linux kernel routine printk(). For debugging, SMSC_TRACE points can temporarily be added anywhere they are needed. SMSC_TRACE points are enabled when USE_TRACE is defined during compile time and ((debug_mode&0x01UL) == 0x01UL) during run time. It is used to indicate general information such as driver parameters used at load time.

For example:

```
int globalDriverParameter=0;

void DriverInitializationEntryPoint()
{
    SMSC_TRACE(“globalDriverParameter = %d”,globalDriverParameter);
    Do other initialization;
}
//When loading a driver in Linux
//  globalDriverParameter could be modified before
//  DriverInitializationEntryPoint is called. Therefore
//  it is useful to display what ever setting has been applied to
//  globalDriverParameter.
```

8.1.2 SMSC_WARNING(message, parameters)

SMSC_WARNING points work just like SMSC_TRACE points, with the exception that they are enabled when USE_WARNING is define during compile time and ((debug_mode&0x02UL)==0x02UL) during run time. The slight difference in enabling allows SMSC_WARNING points to be enabled while SMSC_TRACE points are disabled. SMSC_WARNING points are not intended to indicate general information. They are intended for the purpose of exposing unexpected situations, but only those situations that can be handled without crashing the driver.

For example:

```
void SomeDriverEntryPoint(int parameter)
{
    if (parameter==invalid) {
    SMSC_WARNING(“Invalid Parameter = %d”,parameter);
    return;
}
do work;
}
//if Linux calls an entry point of the driver with
// invalid parameters, issue a warning and
// just return with out any further problems.
```

8.1.3 SMSC_ASSERT(condition)

SMSC_ASSERT points are enabled when USE_ASSERT is defined during compile time. It has no run time dependency on debug mode. This macro is used to test assumptions made when coding. It is useful to ensure consistency of logic through out the driver. It is only intended to be used in situations where a failure of the condition is fatal. That is if code execution were allowed to continue it is assumed that only further unrecoverable errors would occur and so this macro includes an infinite loop to prevent further corruption. Assertion points are also useful when multiple programmers will be maintaining the code. If the original programmer is good at using assertions then any maintainers will be immediately alerted (during testing) if they break an original assumption. Assertions also serve as a good way of documenting assumptions, which helps prevent maintainers from breaking them in the first place.

Examples:

```
SMSC_ASSERT(pointer!=NULL);
SMSC_ASSERT(index<size);
```

8.2 GPIO pins in Conjunction with Oscilloscope

Another method of debugging which can be very helpful is the use of the GPIO pins in conjunction with an Oscilloscope. There are 3 GPIO pins available, which can be configured through the **GPIO_CFG** register. The controller offers two additional output-only GPO pins, shown below as GP_3 and GP_4. The driver is capable of setting these pins high or low, and the resulting waveform can be observed on the scope. Below is a scope trace that illustrates the use of GPIO pins. In this picture NCS, NRD, and NWR are taken from the bus signals. TX_EN is an internal signal brought out on a GPIO pin which the MAC uses to signal the PHY that it is sending data. The signal is high when a packet is being transmitted on the wire. RX_DV is another internal signal brought out that is high when a packet is being received on the wire. The driver sets GP_1 high when it begins receiving a packet, and sets it low when it finishes receiving a packet. Also the driver sets GP_2 high when it begins transmitting a packet, and sets it low when it finishes transmitting a packet. Displaying all these signals in real time on a scope can give a clearer picture of data moving around the system. For example, the GP_2 signal shows that six packets were packed into the transmit FIFO before one packet was received back.

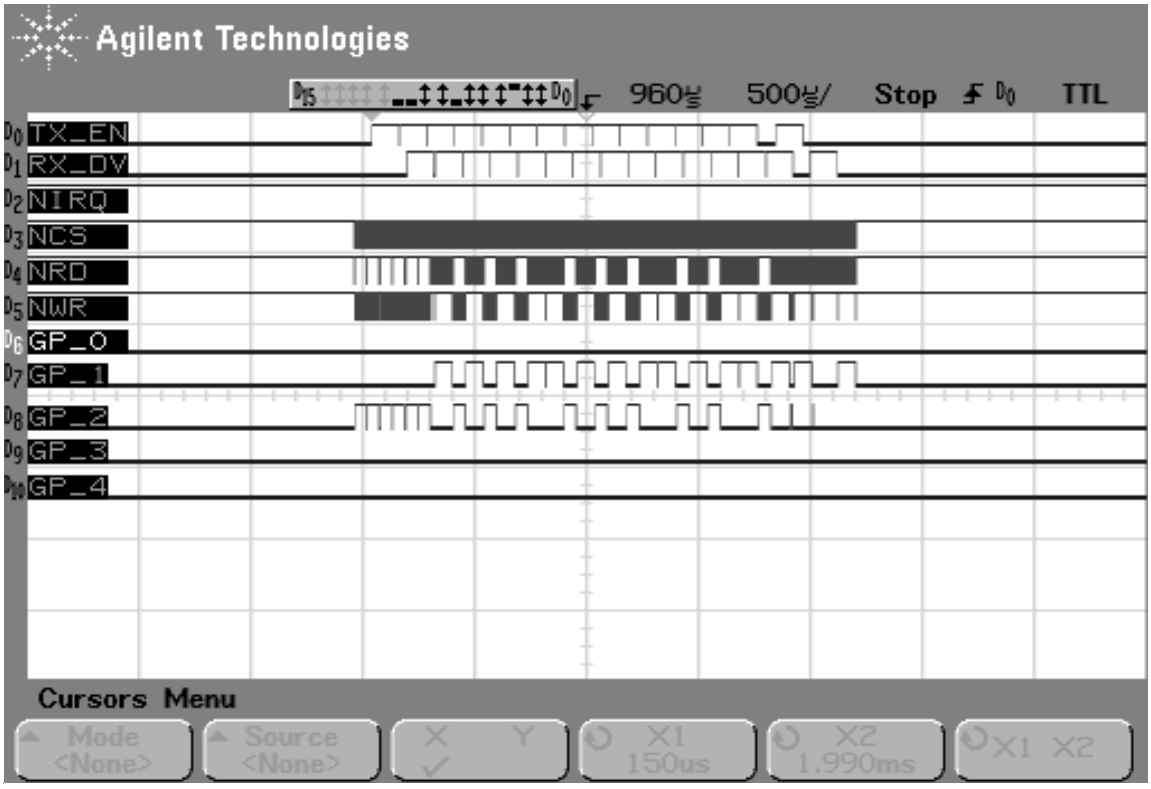

Figure 8.1 Oscilloscope/Logic Analyzer Display

GPIOs can also be used with a scope for instrumentation. Most scopes provide a way to measure the time between events. In the trace above, the scope could be used to measure the time between the rising and falling edges of GP_1, revealing the length of time it takes to read a packet out of the Rx data FIFO with high accuracy. Measurements of the low time could be used to calculate the percentage of bus usage. Almost any timing measurement imaginable can be measured using GPIOs and a scope.

GPIOs can also be used to create trigger points for the scope. If some kind of error is occurring in the driver, it is useful to see what other events are occurring simultaneously. Pulsing the GPIO pin when the error condition occurs provides a convenient sampling point for these events.

8.3 TxCLK and RxCLK

These signals can also be brought out on the GPIO pins to observe TX and RX characteristics. They could be used in conjunction with an inexpensive frequency meter during a manufacture test to measure the data bit rate on the wire. A frequency meter with accuracy of 50 ppm or better is generally required to determine standards compliance.

8.4 Error Interrupts are Invaluable Tools

There are several interrupts available for detecting errors. Some of them can signal when the driver is using the FIFOs incorrectly. They are Receiver Error (RXE), and Transmitter Error (TXE). If the driver makes any mistakes with respect to data alignment, offsets, etc., it will trigger one of these interrupts. When combined with a GPIO trigger pulse or a warning message, these interrupts can contribute greatly to the robustness of the driver, because under normal operation these errors never occur. If they do, then the driver likely has a bug which should be investigated.

Some other useful error interrupts to be aware of include:

- TXSO
- TDFO
- TSFF – TX status FIFO full
- RSFF – RX status FIFO full

8.5 Integrating the Driver: Early Testing

When the driver development has come to the point where it is being integrated into the protocol stack, a basic integration test which is useful for shaking out the data flows is to send the host a stream of ICMP ping request packets, while looking for the ping responses. Once this basic skill is mastered by the driver, the test should advance by varying the packet size through all possible values (64 – 1518 bytes). A PC could be used in this case, although the standard ping program supplied does not automatically change the size of the packets sent. A better choice in this regard would be a packet tester, such as those made by SmartBits or iXia. Continuing the advance, the testing should add faulty packets injected at random points in the ping packet stream. Faulty packets should include, but not be limited to runts, jabbers, and CRC errors.

When the ping responses coming back from the host maintain their correct sequence, this verifies to a large extent that the low-level receive operation is working, that the packets are transferred and aligned into the packet memory buffers correctly, and that the protocol stack can identify the packets correctly as being of type ICMP. It also verifies that the low-level transmit routine basically works, too. Adding in the random faulty packet errors validates the data error processing, as well as showing that the low-level receive operation is not deterred by bad packets. This can be most important when using DMA for receive data transfers, as the driver will need to interleave its “dialogues” with both the device and the DMA engine. At this point in the testing the management features can also be verified by comparing the packet statistics derived from the driver to the values generated by the ping stream request source.

The rate of the ping packets should be well within the abilities of the host hardware. Emphasis should be on the correctness of the response sequence initially, not the data rate of the packet stream. Ideally the test is striving for a data rate that manages to keep at least a few packets in the Rx FIFO while the device is under test, and this can really only be determined by the hardware.

9 Power Management Events

LAN9118 Family members support power-down modes to allow the application to minimize power consumption. These modes operate via the PME Block to control the necessary internal signals for reduced power modes.

9.1 System Description

There is one normal operating power state, D0 and there are two power saving states: D1 and D2. Upon entry into any of the three power management states, only the **PMT_CTRL** register is accessible for read operations. Reads of any other addresses are forbidden until the READY bit is set. All writes, with the exception of the wakeup write to **BYTE_TEST** register, are also forbidden until the READY bit is set. Only when in the D0 (Normal) state, when the READY bit is set, can the rest of the device be accessed.

Entering power saving states involves setting the desired power management state in the **PMT_CTL:PM_MODE** field. If the PM interrupt is enabled, the device can be woken up by writing to the **BYTE_TEST** register.

80 ARKAY DRIVE, HAUPPAUGE, NY 11788 (631) 435-6000, FAX (631) 273-3123