AN2927

LAN743x Programmer's Guide

Author: *Parthiv Pandya*
Microchip Technology Inc.

INTRODUCTION

This document provides basic instructions on how to write a simple driver for the LAN743x. This is written to be operating system (OS) agnostic, and where details are important, C-style pseudo code are provided. It is assumed the user of this document is familiar with writing device drivers for PCIe devices.

Sections

This document includes the following topics:

References

Consult the following documents for details on the specific parts referred to in this document:

- *LAN7430 Data Sheet*
- *LAN7431 Data Sheet*

SYSTEM APIs

This document is intended to be OS agnostic. However, the following pseudo code will need to reference some functions that are usually provided by the OS. Thus, these functions must be defined so that they can be matched properly with other OSs. The following are function definitions that will be referenced in pseudo code throughout this document:

```
/* Some types used in this document
 *     uptr: This is an unsigned integer whose bit width
 *         is the same as the native pointer bit width.
 *         Which may be 32 bit or 64 bit depending on your system.
 *     u32: This is a 32 bit unsigned integer.
 *     u16: This is a 16 bit unsigned integer.
 *     u8: This is an 8 bit unsigned integer.
 */

/* Function: READ
 * Parameters:
 *     uptr address: This is the byte address to read from.
 * Return value:
 *     returns an unsigned 32 bit integer containing
 *     the value read from the location specified by 'address'
 */
u32 READ(uptr address);

/* Function: WRITE
 * Parameters:
 *     uptr address: This is the byte address to write to.
 *     u32 value: This is the 32 bit unsigned integer, whose value
 *         is to be written to the location specified by 'address'
 * Return value: none
 */
void WRITE(uptr address, u32 value);

/* Macro: WAIT
 *     This macro is used to wait until a condition is true.
 *     Since this is described as a macro, condition can include
 *        a function call which will be recalled each time condition is
 *        checked.
 *     Assumed to be always successful.
 * Parameter:
 *     condition is treated as a Boolean.
 */
WAIT(condition);

/* Function: DMA_ADDR_HIGH32, DMA_ADDR_LOW32
 *     A DMA address for a specific memory location may be different than
 *     a CPU address for the same location.
 *     The following functions return the high and low 32 bits of the 64 bit
 *     DMA address converted from the specified cpu pointer address.
 */
u32 DMA_ADDR_HIGH32(void * cpu_addr);
u32 DMA_ADDR_LOW32(void * cpu_addr);

/* Type: isr_type
 *     the isr type
 */
typedef bool (*isr_type)();
```

```
/* Function: REGISTER_ISR
 *     This function is used to register an isr to an irq signal.
 * Parameters:
 *     int irq: The irq number to associate the isr with.
 *     isr_type isr: the isr to be called when the irq has signaled.
 * Return value: none, assumed successful.
 */
void REGISTER_ISR(int irq, isr_type isr);

/* struct packet_buffer: Used to exchange packet data between OS and driver.
 *     u8 * buffer_ptr: points to the first byte of the ethernet frame.
 *     int buffer_length: specifies the length of packet buffer.
 */
struct packet_buffer {
    u8 * buffer_ptr;
    int buffer_length;
}

/* Function: ALLOCATE_PACKET_BUFFER
 *     Allocates a packet buffer. Typically used to prepare a space for
 *     a received packet to go.
 * Parameters:
 *     int length: length of requested buffer.
 * Return value:
 *     pointer to packet_buffer structure, with a valid buffer_ptr, and
 *     a buffer_length equal to length. Assumed successful.
struct packet_data * ALLOCATE_PACKET_BUFFER(int length);

/* Function: TX_COMPLETE
 *     Notifies the OS that the packet has been sent, and the driver
 *     no longer needs to hold the buffer.
 */
void TX_COMPLETE(struct packet_buffer * packet);

/* Function: RX_RECEIVED
 *     Passes a received packet to the OS.
 * Parameters:
 *     struct packet_buffer * packet: pointer to packet data buffer.
 *     int received_length: length of received packet, which may be
 *         shorter than the packet buffer length.
 */
void RX_RECEIVED(struct packet_buffer * packet, int received_length);
```

DEVICE RECOGNITION

Before initializing a device, make sure that the device is present and accessible. The LAN743x is a PCIe device, and assuming it is attached to a PCIe bus, it will first provide access requirements through the PCIe configuration space. The PCIe configuration space is standardized, and OSs usually read this information and reserve memory or input/output (I/O) space for register access needs. To understand the requirements to match your driver to a device visible to it on the bus, read about your OS PCIe driver features.

The Vendor ID (VID) and the Device ID (DID) are important fields in the PCIe configuration space. These are used to associate your driver to the device.

The VID is a 16-bit field found at byte 0 and 1 of the PCIe configuration space. The VID for the LAN743x is 1055h.

The DID is a 16-bit field found at byte 2 and 3 of the PCIe configuration space. The DID for the LAN7430 is 7430h, and the DID for LAN7431 is 7431h.

Other important fields are the Base Address 0 and Base Address 1 (both also referred to as CSR_BASE), which make a 64-bit base address for the control and system registers. This is usually initialized by the OS when memory or I/O space has been reserved for the device. The control and status registers are all 32-bit registers. Most device accesses will be performed through these registers.

To confirm that the correct location is being accessed, it is recommended that the ID-REV register be read first. The ID_REV register is found at (CSR_BASE + 0). The Chip ID field is the high 16 bits of the ID_REV register. The low 16 bits are the Chip Revision.

The Chip ID of LAN7430 is 7430h, and the Chip ID of LAN7431 is 7431h.

The following code is an example of how to verify if your device is accessible:

```
#define ID_REV          (CSR_BASE + 0x0000)
#define CHIP_ID_MASK    (0xFFFF0000)
#define CHIP_ID_7430    (0x74300000)
#define CHIP_ID_7431    (0x74310000)

bool check_chip_id()
{
    u32 id_rev = READ(ID_REV) & CHIP_ID_MASK;

    if((id_rev == CHIP_ID_7430) || (id_rev == CHIP_ID_7431)) {
        /* Device is accessible. */
        return true;
    }
    /* Device is not accessible */
    return false;
}
```

Once the device is confirmed to be accessible, then you can begin the initialization process.

LITE RESET

Resetting the device is the first step. The LAN743x has several different resets available. Soft Lite Reset will be used in this guide. This type of reset does a soft device reset without resetting the PCIe interface so that device access is not lost. The following code performs a soft reset:

```
#define HW_CFG          (CSR_BASE + 0x0010)
#define HW_CFG_LRST_    (0x00000002)

void lite_reset()
{
    u32 temp;

    temp = READ(HW_CFG);
    WRITE(HW_CFG, temp | HW_CFG_LRST_);
    WAIT(!(READ(HW_CFG) & HW_CFG_LRST_));
}
```

INITIALIZATIONS

It is recommended to initialize all the necessary blocks as described in this section.

Media Access Control (MAC)

The MAC must be initialized. Set automatic duplex and speed detection.

```
#define MAC_CR         (CSR_BASE + 0x0100)
#define MAC_CR_ADD_    (0x00001000)
#define MAC_CR_ASD_    (0x00000800)

void mac_set_autodetect()
{
    u32 temp;

    temp = READ(MAC_CR);
    WRITE(MAC_CR, temp | MAC_CR_ADD_ | MAC_CR_ASD_);
}
```

The MAC address is usually stored in the EEPROM or One-Time Programmable (OTP). Assuming it is attached and properly programmed, the LAN743x will read it on power up. The MAC address will be available to read from the MAC_RX_ADDRH and MAC_RX_ADDRL registers. The following functions show how to read and write the MAC address:

```
#define MAC_RX_ADDRH   (CSR_BASE + 0x0118)
#define MAC_RX_ADDRL   (CSR_BASE + 0x011C)

void mac_get_address(u8 * mac_address)
{
    /* Assuming mac_address is a pointer to an array of 6 u8 types */
    u32 mac_rx_addrh;
    u32 mac_rx_addrl;

    mac_rx_addrh = READ(MAC_RX_ADDRH);
    mac_rx_addrl = READ(MAC_RX_ADDRL);
    mac_address[0] = mac_rx_addrl & 0xFF;
    mac_address[1] = (mac_rx_addrl >> 8) & 0xFF;
    mac_address[2] = (mac_rx_addrl >> 16) & 0xFF;
    mac_address[3] = (mac_rx_addrl >> 24) & 0xFF;
    mac_address[4] = (mac_rx_addrh & 0xFF;
    mac_address[5] = (mac_rx_addrh >> 8) & 0xFF;
}

void mac_set_address(u8 * mac_address)
{
    /* Assuming mac_address is a pointer to an array of 6 u8 types */
    u32 mac_rx_addrh;
    u32 mac_rx_addrl;

    mac_rx_addrl = mac_address[0] |
                   mac_address[1] << 8 |
                   mac_address[2] << 16 |
                   mac_address[3] << 24;
    mac_rx_addrh = mac_address[4] |
                   mac_address[5] << 8;
    WRITE(MAC_RX_ADDRH, mac_rx_addrh);
    WRITE(MAC_RX_ADDRL, mac_rx_addrl);
}
```

Receive Filtering Engine (RFE)

The MAC_RX_ADDRH and MAC_RX_ADDRL registers are not used for packet filtering. By default, all unicast and multicast packets are received, and broadcast packets are blocked. The following functions can be used to change the filtering settings:

```
#define ADDR_FILT_HI(x)       (CSR_BASE + 0x400 + (8 * (x)))
#define ADDR_FILT_HI_VALID_   (0x80000000)
#define ADDR_FILT_LO(x)       (CSR_BASE + 0x404 + (8 * (x)))

#define RFE_CTL               (CSR_BASE + 0x0508)
#define RFE_CTL_AB_           (0x00000400)
#define RFE_CTL_DPF_          (0x00000002)

void set_perfect_filter_address(int index, u8 * mac_address)
{
    /* index is a number between 0 and 32
     *     0 is usually used for the local mac address
     *     1 to 32 can be used for multicast addresses
     *     index can be 32, as there are 33 table entries.
     * mac_address is the address to assign to the
     *     perfect filter table entry at 'index'
     *     it is a pointer to an array of 6 u8 types.
     */
    u32 addr_filt_hi;
    u32 addr_filt_lo;

    /* clear valid bit so hardware does not use this
     *     entry before its valid again.
     */
    WRITE(ADDR_FILT_HI(index), 0);

    addr_filt_lo = mac_address[0] |
                   (mac_address[1] << 8) |
                   (mac_address[2] << 16) |
                   (mac_address[3] << 24);
    addr_filt_hi = mac_address[4] |
                   (mac_address[5] << 8) |
                   ADDR_FILT_HI_VALID_;
    WRITE(ADDR_FILT_LO(index), addr_filt_lo);
    WRITE(ADDR_FILT_HI(index), addr_filt_hi);
}

void set_receive_broadcast(bool enable_broadcast)
{
    u32 rfe_ctl;

    rfe_ctl = READ(RFE_CTL);
    if(enable_broadcast)
        rfe_ctl |= RFE_CTL_AB_;
    else
        rfe_ctl &= ~RFE_CTL_AB_;
    WRITE(RFE_CTL, rfe_ctl);
}

void set_perfect_filter_enable(bool enable_perfect_filter)
{
    u32 rfe_ctl;

    rfe_ctl = READ(RFE_CTL);
    if(enable_perfect_filter)
        rfe_ctl |= RFE_CTL_DPF_;
    else
        rfe_ctl &= ~RFE_CTL_DPF_;
    WRITE(RFE_CTL, rfe_ctl);
}
```

The LAN743x also supports hash filtering, but that is outside the scope of this document. Refer to the *LAN7430/ LAN7431 Data Sheet* for more information.

To prevent receiving all packets, the perfect filter must be enabled and the local MAC address must be added to the perfect filter table. The receiving broadcast must also be enabled. The following is a simple RFE initializer filter function:

```
void rfe_initialize()
{
    u8 local_mac_address[6];

    mac_get_address(local_mac_address);
    set_perfect_filter_address(0, local_mac_address);
    set_perfect_filter_enable(true);
    set_receive_broadcast(true);
}
```

PHY

Before using the PHY, perform reset as follows:

```
#define PMT_CTL                (CSR_BASE + 0x0014)
#define PMT_CTL_ETH_PHY_RST_   (0x00000010)
#define PMT_CTL_READY_         (0x00000080)

void phy_reset()
{
    u32 temp;

    temp = READ(PMT_CTL);
    WRITE(PMT_CTL, temp | PMT_CTL_ETH_PHY_RST_);

    WAIT(!(READ(PMT_CTL) & PMT_CTL_ETH_PHY_RST_));
    WAIT(READ(PMT_CTL) & PMT_CTL_READY_);
}
```

After the PHY has been reset, the PHY registers can be accessed with the following functions:

```
#define MII_ACCESS                    (CSR_BASE + 0x0120)
#define MII_ACCESS_PHY_ADDR_SHIFT_    (11)
#define MII_ACCESS_PHY_ADDR_MASK_     (0x0000F800)
#define MII_ACCESS_REG_INDEX_SHIFT_   (6)
#define MII_ACCESS_REG_INDEX_MASK_    (0x000007C0)
#define MII_ACCESS_MII_READ_          (0x00000000)
#define MII_ACCESS_MII_WRITE_         (0x00000002)
#define MII_ACCESS_MII_BUSY_          (0x00000001)
#define MII_DATA                      (CSR_BASE + 0x0124)

/* phy_read:
 *     returns value of phy register
 */
int phy_read(int phy_addr, int index)
{
    /* phy_addr: for LAN7430, must be 1
     *     for LAN7431 must be address of external phy
     * index: phy register index to read
     */
    u32 mii_access;
```

```
    WAIT(!(READ(MII_ACCESS) & MII_ACCESS_MII_BUSY_));

    mii_access = (phy_addr << MII_ACCESS_PHY_ADDR_SHIFT_) &
                 MII_ACCESS_PHY_ADDR_MASK_;
    mii_access |= (index << MII_ACCESS_REG_INDEX_SHIFT_) &
                  MII_ACCESS_REG_INDEX_MASK_;
    mii_access |= MII_ACCESS_MII_READ_;
    mii_access |= MII_ACCESS_BUSY_;
    WRITE(MII_ACCESS, mii_access);

    WAIT(!(READ(MII_ACCESS) & MII_ACCESS_MII_BUSY_));

    return (int)(READ(MII_DATA) & 0xFFFF);
}
/* phy_write: Write reg_value to phy register
 */
void phy_write(int phy_addr, int index, u16 reg_value)
{
    /* phy_addr: for LAN7430, must be 1
     *     for LAN7431 must be address of external phy
     * index: phy register index to write
     * reg_value: the value to write to the register.
     */
    u32 mii_access;

    WAIT(!(READ(MII_ACCESS) & MII_ACCESS_MII_BUSY_));

    WRITE(MII_DATA, (u32)reg_value);

    mii_access = (phy_addr << MII_ACCESS_PHY_ADDR_SHIFT_) &
                 MII_ACCESS_PHY_ADDR_MASK_;
    mii_access |= (index << MII_ACCESS_REG_INDEX_SHIFT_) &
                  MII_ACCESS_REG_INDEX_MASK_;
    mii_access |= MII_ACCESS_MII_WRITE_;
    mii_access |= MII_ACCESS_BUSY_;
    WRITE(MII_ACCESS, mii_access);

    WAIT(!(READ(MII_ACCESS) & MII_ACCESS_MII_BUSY_));
}
```

With PHY read/write access, a link auto-negotiation can be initiated as follows:

```
#define PHY_ADDR              (1)
/* PHY_ADDR may be different for LAN7431 using external phy */
#define PHY_CTRL              (0)
#define PHY_CTRL_AN_EN_       (0x1000)
#define PHY_CTRL_RST_AN_      (0x0200)

void autonegotiate_link()
{
    phy_write(PHY_ADDR, PHY_CTRL,
        PHY_CTRL_AN_EN_ | PHY_CTRL_RST_AN_);
}
```

Interrupts

The LAN743x supports MSI and MSI-X interrupts, but these are beyond the scope of this guide. For more information on these interrupts, see the *LAN7430/LAN7431 Data Sheet*. This document assumes the use of legacy interrupts.

An Interrupt Service Routine (ISR) is a function that is called when the device has work to be done. The device sends a physical interrupt request (IRQ) signal when it wants its driver to handle something. The IRQ number is system-architecture-dependent, and it is assumed in this guide that the user can obtain it. Pseudo code examples will refer to it as IRQ.

The following are some register definitions and basic interrupt initializer:

```
#define INT_STS        (CSR_BASE + 0x0780)
#define INT_SET        (CSR_BASE + 0x0784)
#define INT_EN_SET     (CSR_BASE + 0x0788)
#define INT_EN_CLR     (CSR_BASE + 0x078C)
#define INT_BIT_RX0_   (0x01000000)
#define INT_BIT_TX0_   (0x00010000)
#define INT_BIT_MAS_   (0x00000001)

bool isr_entry_point();/* defined below */
bool test_interrupt();/* defined below */

bool initialize_interrupts()
{
    /* clear all interrupt enables */
    WRITE(INT_EN_CLR, 0xFFFFFFFF);

    /* clear any existing interrupt status */
    WRITE(INT_STS, 0xFFFFFFFF);

    /* register the isr to the IRQ */
    REGISTER_ISR(IRQ, isr_entry_point);

    /* Enable master interrupt */
    WRITE(INT_EN_SET, INT_BIT_MAS_);

    /* specific interrupts will be enabled later
     *     when specific blocks are initialized.
     */

    if (!test_interrupt())
        /* interrupt test failed, report error */
        return false;

    /* interrupts initialized successfully */
    return true;
}
```

The following is a basic ISR entry point:

```
bool isr_entry_point()
{
    u32 int_sts;
    u32 int_en;

    /* read interrupt status */
    int_sts = READ(INT_STS);

    /* read interrupt enable */
    int_en = READ(INT_EN_SET);

    int_sts &= int_en;
    if (!(int_sts & INT_BIT_MAS_))
        /* master bit not set, can't be ours */
        return false;
```

```
        if(int_sts) {
            /* one of our enabled interrupts has signaled */

            test_isr(int_sts);/* defined below */

            tx_isr(int_sts);/* defined later */

            rx_isr(int_sts);/* defined later */

            /* return true, interrupt recognized. */
            return true;
        }
        /* return false, interrupt not recognized. */
        return false;
    }
```

Testing the ISR after registration is recommended. The following code illustrates a test mechanism, which was referenced in Interrupts on page 9.

```
#define INT_BIT_SW_GP_         (0x00000200)

bool isr_test_flag = false;

void test_isr(u32 int_sts)
{
    if (int_sts & INT_BIT_SW_GP_) {
        isr_test_flag = true;

        /* clear interrupt status */
        WRITE(INT_STS, INT_BIT_SW_GP_);
    }
}

bool test_interrupts()
{
    /* assuming isr is already registered
     *     and the master interrupt enable is set.
     */
    isr_test_flag = false;

    /* clear status */
    WRITE(INT_STS, INT_BIT_SW_GP_);

    /* enable software interrupt */
    WRITE(INT_EN_SET, INT_BIT_SW_GP_);

    /* activate software interrupt */
    WRITE(INT_SET, INT_BIT_SW_GP_);

    WAIT(isr_test_flag);
    /* assuming the previous WAIT includes a timeout in case the test failed. *

    /* disable software interrupt */
    WRITE(INT_EN_CLR, INT_BIT_SW_GP_);

    /* clear status */
    WRITE(INT_STS, INT_BIT_SW_GP_);

    return isr_test_flag;
}
```

Direct Memory Access (DMA) Controller

The DMA controller is used to transfer packet data between the system memory and the LAN743x. The LAN743x has four receive (RX) channels and one transmit (TX) channel. The register and function definitions below are written to support multiple RX and TX channels. However, this document focuses on RX channel 0 and TX channel 0. For more information on the requirements to support multiple RX channels, refer to the *LAN7430/LAN7431 Data Sheet*.

The DMAC_CMD register has several useful commands for resetting, starting, and stopping DMAC channels. The following functions show how to use these commands:

```
#define DMAC_CMD                    (CSR_BASE + 0xC0C)
#define DMAC_CMD_SWR_               (0x80000000)
#define DMAC_CMD_TX_SWR_(channel)   (1 << (24 + (channel))
#define DMAC_CMD_START_T_(channel)  (1 << (20 + (channel))
#define DMAC_CMD_STOP_T_(channel)   (1 << (16 + (channel))
#define DMAC_CMD_RX_SWR_(channel)   (1 << (8 + (channel))
#define DMAC_CMD_START_R_(channel)  (1 << (4 + (channel))
#define DMAC_CMD_STOP_R_(channel)   (1 << (0 + (channel))

#define DMAC_INT_STS                (CSR_BASE + 0xC10)
#define DMAC_INT_EN_SET             (CSR_BASE + 0xC14)
#define DMAC_INT_EN_CLR             (CSR_BASE + 0xC18)
#define DMAC_INT_BIT_RX0_           (0x00010000)
#define DMAC_INT_BIT_TX0_           (0x00000001)

/* dmac_reset: This is a full DMAC reset. All RX and TX channels are affected.
 * do this only during initialization, any transfers in progress will be aborted.
 */
void dmac_reset()
{
    WRITE(DMAC_CMD, DMAC_CMD_SWR_);
    WAIT(!(READ(DMAC_CMD) & DMAC_CMD_SWR_));
}

/* dmac_tx_reset: This is will reset an individual tx channel
 * NOTE: LAN743x only supports tx channel 0
 */
void dmac_tx_reset(int channel)
{
    u32 tx_reset_bit = DMAC_CMD_TX_SWR_(channel);

    WRITE(DMAC_CMD, tx_reset_bit);
    WAIT(!(READ(DMAC_CMD) & tx_reset_bit));
}

/* dmac_rx_reset: This is will reset an individual rx channel,
 * NOTE: only channels 0, 1, 2, and 3 are supported
 */
void dmac_rx_reset(int channel)
{
    u32 rx_reset_bit = DMAC_CMD_RX_SWR_(channel);

    WRITE(DMAC_CMD, rx_reset_bit);
    WAIT(!(READ(DMAC_CMD) & rx_reset_bit));
}
#define DMAC_CHANNEL_STATE_SET(start_bit, stop_bit) \
    (((start_bit) ? 2 : 0) | ((stop_bit) ? 1 : 0))
#define DMAC_CHANNEL_STATE_INITIAL          DMAC_CHANNEL_STATE_SET(0, 0)
#define DMAC_CHANNEL_STATE_STARTED          DMAC_CHANNEL_STATE_SET(1, 0)
#define DMAC_CHANNEL_STATE_STOP_PENDING     DMAC_CHANNEL_STATE_SET(1, 1)
#define DMAC_CHANNEL_STATE_STOPPED          DMAC_CHANNEL_STATE_SET(0, 1)
```

```
int dmac_tx_get_state(int channel)
{
    u32 dmac_cmd = READ(DMAC_CMD);
    return DMAC_CHANNEL_STATE_SET((dmac_cmd & DMAC_CMD_START_T_(channel)),
                                  (dmac_cmd & DMAC_CMD_STOP_T_(channel)));
}

/* dmac_tx_start: starts the specified Tx DMA channel
 * NOTE: must have already initialized the Tx DMA ring
 */
void dmac_tx_start(int channel)
{
    /* don't attempt to start if stop is pending */
    WAIT(dmac_tx_get_state(channel) != DMAC_CHANNEL_STATE_STOP_PENDING);

    WRITE(DMAC_CMD, DMAC_CMD_START_T_(channel));
}

/* dmac_tx_stop: stops the specified Tx DMA channel
 *   do this before cleaning up resources shared with the LAN743x.
 */
bool dmac_tx_stop(int channel)
{
    WRITE(DMAC_CMD, DMAC_CMD_STOP_T_(channel);
    WAIT(dmac_tx_get_state(channel) != DMAC_CHANNEL_STATE_STOP_PENDING);
}
int dmac_rx_get_state(int channel)
{
    u32 dmac_cmd = READ(DMAC_CMD);
    return DMAC_CHANNEL_STATE_SET((dmac_cmd & DMAC_CMD_START_R_(channel)),
                                  (dmac_cmd & DMAC_CMD_STOP_R_(channel)));
}

/* dmac_rx_start: starts the specified Rx DMA channel
 * NOTE: must have already initialized the Rx DMA ring
 */
void dmac_rx_start(int channel)
{
    /* don't attempt to start if stop is pending */
    WAIT(dmac_rx_get_state(channel) != DMAC_CHANNEL_STATE_STOP_PENDING);

    WRITE(DMAC_CMD, DMAC_CMD_START_R_(channel));
}

/* dmac_rx_stop: stops the specified Rx DMA channel
 *   do this before cleaning up resources shared with the LAN743x.
 */
void dmac_rx_stop(int channel)
{
    WRITE(DMAC_CMD, DMAC_CMD_STOP_R_(channel);
    WAIT(dmac_rx_get_state(channel) != DMAC_CHANNEL_STATE_STOP_PENDING);
}
```

First-In, First-Out (FIFO) Controller

The FIFO controller exchanges packet data with the DMA controller. There is an individual FIFO for each RX channel and one for the single TX channel. The following functions show how to reset, enable, and disable individual FIFO:

```
#define FCT_RX_CTL                  (CSR_BASE + 0xAC)
#define FCT_RX_CTL_EN_(channel)     (1 << (28 + (channel)))
#define FCT_RX_CTL_DIS_(channel)    (1 << (24 + (channel)))
#define FCT_RX_CTL_RESET_(channel)  (1 << (20 + (channel)))

void fct_rx_reset(int channel)
{
    WRITE(FCT_RX_CTL, FCT_RX_CTL_RESET_(channel));
    WAIT(!(READ(FCT_RX_CTL) & FCT_RX_CTL_RESET_(channel)));
}

void fct_rx_enable(int channel)
{
    WRITE(FCT_RX_CTL, FCT_RX_CTL_EN_(channel));
}

void fct_rx_disable(int channel)
{
    WRITE(FCT_RX_CTL, FCT_RX_CTL_DIS_(channel));
    WAIT(!(READ(FCT_RX_CTL) & FCT_RX_CTL_EN_(channel));
}

#define FCT_TX_CTL                  (CSR_BASE + 0xC4)
#define FCT_TX_CTL_EN_(channel)     (1 << (28 + (channel)))
#define FCT_TX_CTL_DIS_(channel)    (1 << (24 + (channel)))
#define FCT_TX_CTL_RESET_(channel)  (1 << (20 + (channel)))

void fct_tx_reset(int channel)
{
    WRITE(FCT_TX_CTL, FCT_TX_CTL_RESET_(channel));
    WAIT(!(READ(FCT_TX_CTL) & FCT_TX_CTL_RESET_(channel)));
}

void fct_tx_enable(int channel)
{
    WRITE(FCT_TX_CTL, FCT_TX_CTL_EN_(channel));
}

void fct_tx_disable(int channel)
{
    WRITE(FCT_TX_CTL, FCT_TX_CTL_DIS_(channel));
    WAIT(!(READ(FCT_TX_CTL) & FCT_TX_CTL_EN_(channel));
}
```

TX DMA Ring

The TX DMA Ring is an array of data structures that is understood by the LAN743x. It tells the LAN743x where to find in the system memory a packet to transmit. The following code shows how to set up the TX DMA Ring:

```
#define TX_RING_SIZE                        (16)
#define TX_CFG_A(channel)                   (CSR_BASE + 0xD40 + ((channel) << 6))
#define TX_CFG_A_HP_WB_EN_                  (0x00000020)
#define TX_CFG_A_TX_TMR_HPWB_SEL_IOC_       (0x10000000)
#define TX_CFG_B(channel)                   (CSR_BASE + 0xD44 + ((channel) << 6))
#define TX_CFG_B_TX_RING_LEN_MASK_          (0x0000FFFF)
#define TX_BASE_ADDRH(channel)              (CSR_BASE + 0xD48 + ((channel) << 6))
#define TX_BASE_ADDRL(channel)              (CSR_BASE + 0xD4C + ((channel) << 6))
#define TX_HEAD_WB_ADDRH(channel)           (CSR_BASE + 0xD50 + ((channel) << 6))
#define TX_HEAD_WB_ADDRL(channel)           (CSR_BASE + 0xD54 + ((channel) << 6))
#define TX_HEAD(channel)                    (CSR_BASE + 0xD58 + ((channel) << 6))
#define TX_TAIL(channel)                    (CSR_BASE + 0xD5C + ((channel) << 6))

struct tx_descriptor {
    u32 data0;
    u32 data1;
    u32 data2;
    u32 data3;
};

#define TX_DESC_DATA0_FS_                   (0x20000000)
#define TX_DESC_DATA0_LS_                   (0x10000000)
#define TX_DESC_DATA0_IOC_                  (0x04000000)
#define TX_DESC_DATA0_FCS_                  (0x00020000)
#define TX_DESC_DATA0_BUF_LENGTH_MASK_      (0x0000FFFF)
#define TX_DESC_DATA3_FRAME_LENGTH_MASK_    (0x3FFF0000)

/* tx_descriptor_ring: This is the descriptor ring which will be accessed by the
 *     LAN743x. For simplicity we allocate it here in global space. But a typical
 *     driver may need to allocate it with special DMA accessibility privileges.
 */
struct tx_descriptor tx_descriptor_ring[TX_RING_SIZE];

/* tx_buffer_ring: This is just a place to hold the packet buffers until the LAN743x
 *     indicates completion.
 */
struct packet_buffer * tx_buffer_ring[TX_RING_SIZE];

/* tx_head_write_back: This is used by the LAN743x to write the head index
 *     back to system memory.
 */
u32 tx_head_write_back;
u32 tx_last_head;/* stores the last known head index. */
u32 tx_last_tail;/* stores the last tail index. */

/* tx_ring_setup: Used to initialize the TX Ring */
void tx_ring_setup()
{
    u32 temp;

    memset(&tx_descriptor_ring, 0, sizeof(tx_descriptor_ring));
    memset(&tx_buffer_ring, 0, sizeof(tx_buffer_ring));
    tx_head_write_back = 0;

    fct_tx_reset(0);
    fct_tx_enable(0);

    dmac_tx_reset(0);
```

```
    /* set descriptor ring base address */
    WRITE(TX_BASE_ADDRH(0), DMA_ADDR_HIGH32(&(tx_descriptor_ring[0])));
    WRITE(TX_BASE_ADDRL(0), DMA_ADDR_LOW32(&(tx_descriptor_ring[0])));

    /* set ring size */
    temp = READ(TX_CFG_B(0));
    temp &= ~TX_CFG_B_TX_RING_LEN_MASK_;
    temp |= (TX_RING_SIZE & TX_CFG_B_TX_RING_LEN_MASK_);
    WRITE(TX_CFG_B(0), temp);

    /* Enable interrupt on completion and head pointer writeback */
    temp = TX_CFG_A_TX_TMR_HPWB_SEL_IOC_ | TX_CFG_A_TX_HP_WB_EN_;
    WRITE(TX_CFG_A(0), temp);

    /* set head pointer write back address */
    WRITE(TX_HEAD_WB_ADDRH(0), DMA_ADDR_HIGH32(&tx_head_write_back));
    WRITE(TX_HEAD_WB_ADDRL(0), DMA_ADDR_LOW32(&tx_head_write_back));

    /* due to previous reset, tx_last_head should be 0 after this READ */
    tx_last_head = READ(TX_HEAD(0));

    /* set tail to 0 */
    tx_last_tail = 0;
    WRITE(TX_TAIL(0), tx_last_tail);

    /* with head and tail at 0, that means the ring is empty */

    /* enable interrupts */
    WRITE(INT_EN_SET, INT_BIT_TX0_);
    WRITE(DMAC_INT_EN_SET, DMAC_INT_BIT_TX0_);

    dmac_tx_start(0);
}
```

RX DMA Ring

The RX DMA Ring is an array of data structures that understood by the LAN743x. It tells the LAN743x where to store in the system memory a received packet. The following code shows how to set up an RX DMA Ring:

```
#define RX_RING_SIZE                (16)
#define RX_CFG_A(channel)           (CSR_BASE + 0xC40 + ((channel) << 6))
#define RX_CFG_A_RX_HP_WB_EN_       (0x00000020)
#define RX_CFG_B(channel)           (CSR_BASE + 0xC44 + ((channel) << 6))
#define RX_CFG_B_RX_PAD_MASK_       (0x03000000)
#define RX_CFG_B_RX_PAD_0_          (0x00000000)
#define RX_CFG_B_RX_PAD_2_          (0x02000000)
#define RX_CFG_B_RX_RING_LEN_MASK_  (0x0000FFFF)
#define RX_BASE_ADDRH(channel)      (CSR_BASE + 0xC48 + ((channel) << 6))
#define RX_BASE_ADDRL(channel)      (CSR_BASE + 0xC4C + ((channel) << 6))
#define RX_HEAD_WB_ADDRH(channel)   (CSR_BASE + 0xC50 + ((channel) << 6))
#define RX_HEAD_WB_ADDRL(channel)   (CSR_BASE + 0xC54 + ((channel) << 6))
#define RX_HEAD(channel)            (CSR_BASE + 0xC58 + ((channel) << 6))
#define RX_TAIL(channel)            (CSR_BASE + 0xC5C + ((channel) << 6))

struct rx_descriptor {
    u32 data0;
    u32 data1;
    u32 data2;
    u32 data3;
};

/* When OWN bit is set, descriptor is owned by LAN743x */
#define RX_DESC_DATA0_OWN_                  (0x00008000)
/* When OWN bit is clear, descriptor is owned by host */
#define RX_DESC_DATA0_FS_                   (0x80000000)
#define RX_DESC_DATA0_LS_                   (0x40000000)
#define RX_DESC_DATA0_FRAME_LENGTH_MASK_    (0x3FFF0000);
#define RX_DESC_DATA0_FRAME_LENGTH_GET_(data0)\
        (((data0) & RX_DESC_DATA0_FRAME_LENGTH_MASK_) >> 16)
#define RX_DESC_DATA0_BUF_LENGTH_MASK_      (0x00003FFF)

/* rx_descriptor_ring: This is a descriptor ring which will be accessed by the
 *     LAN743x. For simplicity we allocate it here in global space. But a typical
 *     driver may need to allocate it with special DMA accessibility privileges.
 */
struct rx_descriptor rx_descriptor_ring[RX_RING_SIZE];

/* rx_buffer_ring: This is just a place to hold the packet buffers until they
 *     are ready to be passed to the OS.
 */
struct packet_buffer * rx_buffer_ring[RX_RING_SIZE];

/* rx_head_write_back: Used by the LAN743x to write the head index back to
 *     system memory.
 */
u32 rx_head_write_back;
u32 rx_last_head;
u32 rx_last_tail;

void rx_prepare_ring_element(int index)
{
    struct packet_buffer * packet = ALLOCATE_PACKET_BUFFER(1600);
    rx_buffer_ring[index] = packet;
    rx_descriptor_ring[index].data1 = DMA_ADDR_LOW32(packet->buffer_ptr);
    rx_descriptor_ring[index].data2 = DMA_ADDR_HIGH32(packet->buffer_ptr);
    rx_descriptor_ring[index].data3 = 0;
    rx_descriptor_ring[index].data0 = RX_DESC_DATA0_OWN_ |
        (packet->buffer_length & RX_DESC_DATA0_BUF_LENGTH_MASK_);
}
```

```
/* rx_ring_setup: Used to initialize the RX Ring */
void rx_ring_setup()
{
    int index;
    u32 temp;

    memset(&rx_descriptor_ring, 0, sizeof(rx_descriptor_ring));
    memset(&rx_buffer_ring, 0, sizeof(rx_buffer_ring));
    rx_head_write_back = 0;
    for(index = 0; index < RX_RING_SIZE; index++) {
        rx_prepare_ring_element(index);
    }
    dmac_rx_reset(0);

    /* set ring base address */
    WRITE(RX_BASE_ADDRH(0), DMA_ADDR_HIGH32(&(rx_descriptor_ring[0])));
    WRITE(RX_BASE_ADDRL(0), DMA_ADDR_LOW32(&(rx_descriptor_ring[0])));

    /* set head write back address */
    WRITE(RX_HEAD_WB_ADDRH(0), DMA_ADDR_HIGH32(&rx_head_write_back));
    WRITE(RX_HEAD_WB_ADDRL(0), DMA_ADDR_LOW32(&rx_head_write_back));

    WRITE(RX_CFG_A, RX_CFG_A_RX_HP_WB_EN_);

    temp = READ(RX_CFG_B);
    temp &= ~RX_CFG_B_RX_PAD_MASK_;
    temp |= RX_CFG_B_RX_PAD_0_;
    /* NOTE: use RX_CFG_B_RX_PAD_2_ instead if you want each packet to have
     *    a 2 byte padding at the beginning of the buffer. This allows the
     *    IP header to have better alignment.
     */
    temp &= ~RX_CFG_B_RX_RING_LEN_MASK_;
    temp |= (RX_RING_SIZE & RX_CFG_B_RX_RING_LEN_MASK_);
    WRITE(RX_CFG_B, temp);

    /* set tail and head so all descriptors belong to the LAN743x */
    rx_last_tail = ((u32)(RX_RING_SIZE - 1));
    WRITE(RX_TAIL(0), rx_last_tail);
    rx_last_head = READ(RX_HEAD(0));

    WRITE(INT_EN_SET, INT_BIT_RX0_);
    WRITE(DMAC_INT_STS, DMAC_INT_BIT_RX0_);
    WRITE(DMAC_INT_EN_SET, DMAC_INT_BIT_RX0_);

    dmac_rx_start(0);
    fct_rx_reset(0);
    fct_rx_enable(0);
}
```

Once all initializations have been completed, run-time processing can then be implemented.

TRANSMIT PROCESSING

The following code shows how to send packets:

```
int tx_next_ring_index(int index)
{
    return ((++index) % TX_RING_SIZE);
}

int tx_get_available_ring_space()
{
    int result = tx_last_head - tx_last_tail - 1;

    if (result < 0) result += TX_RING_SIZE;

    return result;
}

/* tx_send_packet: Adds a packet to the tx ring to be transmitted
 *      as soon as possible.
 * Assuming non-reentrant.
 * return true if packet added to ring successfully
 * return false if ring is full.
 */
bool tx_send_packet(struct packet_buffer * packet)
{
    struct tx_descriptor * tx_desc;

    if(tx_get_available_ring_space() <= 0)
        return false;

    tx_desc = &(tx_descriptor_ring[tx_last_tail]);

    tx_desc->data0 = ((packet->buffer_length) & TX_DESC_DATA0_BUF_LENGTH_MASK_) |
                     TX_DESC_DATA0_FS_ | TX_DESC_DATA0_LS_ |
                     TX_DESC_DATA0_IOC_ | TX_DESC_DATA0_FCS;
    tx_desc->data1 = DMA_ADDR_LOW32(packet->buffer_ptr);
    tx_desc->data2 = DMA_ADDR_HIGH32(packet->buffer_ptr);
    tx_desc->data3 = (packet->buffer_length << 16) &
        TX_DESC_DATA3_FRAME_LENGTH_MASK_;

    /* hold packet buffer since it will be accessed by LAN743x */
    tx_buffer_ring[tx_last_tail] = packet;

    /* move tail forward */
    tx_last_tail = tx_next_ring_index(tx_last_tail);

    /* updating the tail register causes our new descriptor to be loaded. */
    /* which will also cause the packet buffer to be loaded into the TX FIFO. */
    WRITE(TX_TAIL(0), tx_last_tail);

    return true;
}
```

The following “tx_isr” shows how to clean up packet buffers when they are no longer needed:

```
/* tx_isr: This is called when a packet buffer has been transferred to
 *     the LAN743x, and is ready to be cleaned up.
 *     You may want to move some of this processing outside the isr.
 */
void tx_isr(u32 int_sts)
{
    if(int_sts & INT_BIT_TX0) {
        /* disable interrupt */
        WRITE(INT_EN_CLR, INT_BIT_TX0_);

        /* clear interrupt status bit */
        WRITE(INT_STS, INT_BIT_TX0_);

        /* clean up descriptors and release packet buffers */
        while(tx_head_write_back != last_head) {
            struct tx_descriptor * tx_desc = tx_descriptor_ring[tx_last_head];
            struct packet_buffer * packet = tx_buffer_ring[tx_last_head];

            tx_desc->data0 = 0;
            tx_desc->data1 = 0;
            tx_desc->data2 = 0;
            tx_desc->data3 = 0;
            tx_buffer_ring[tx_last_head] = NULL;
            TX_COMPLETE(packet);

            /* move head forward */
            tx_last_head = tx_next_ring_index(tx_last_head);
        }

        /* re-enable interrupt */
        WRITE(INT_EN_SET, INT_BIT_TX0_);
    }
}
```

RECEIVE PROCESSING

The following code shows how to process received packets:

```
int rx_next_ring_index(int index)
{
    return ((++index) % RX_RING_SIZE);
}

void rx_isr(u32 int_sts)
{
    int current_head;

    if(int_sts & INT_BIT_RX0) {
        WRITE(INT_EN_CLR, INT_BIT_RX0_);
        WRITE(INT_STS, INT_BIT_RX0_);

        /* NOTE: you may want to process rx packets outside of the isr */
        while((current_head = rx_head_write_back) != rx_last_head)
        {
            struct rx_descriptor * descriptor = &(rx_descriptor_ring[rx_last_head]);
            struct packet_buffer * packet = &(rx_buffer_ring[rx_last_head]);
            int received_length;

            /* make sure OWN bit is not set, meaning driver owns this descriptor */
            if (descriptor->data0 & RX_DESC_DATA0_OWN_)
                break;

            /* make sure FS and LS bits are set,
             *     meaning this is a single buffer packet
             * See data sheet for info on how to support multi buffer packets.
             */
            if (!(descriptor->data0 & RX_DESC_DATA0_FS_))
                break;
            if (!(descriptor->data0 & RX_DESC_DATA0_LS_))
                break;

            /* save received length */
            received_length = RX_DESC_DATA0_FRAME_LENGTH_GET_(descriptor->data0);

            /* clear descriptor and packet buffer pointer */
            memset(descriptor, 0, sizeof(*descriptor));
            rx_buffer_ring[rx_last_head] = NULL;

            /* pass packet to OS */
            RX_RECEIVED(packet, received_length);

            /* prepare ring element for next time */
            rx_prepare_ring_element(rx_last_head);

            /* move head and tail forward */
            rx_last_tail = rx_last_head;
            rx_last_head = rx_next_ring_index(rx_last_head);
        }

        WRITE(INT_EN_SET, INT_BIT_RX0_);

        WRITE(RX_TAIL(0), rx_last_tail);
    }
}
```

APPENDIX A: APPLICATION NOTE REVISION HISTORY

TABLE A-1: REVISION HISTORY

Revision Level & Date	Section/Figure/Entry	Correction
DS00002927A (01-23-19)	Initial release.	

THE MICROCHIP WEB SITE

Microchip provides online support via our WWW site at www.microchip.com. This web site is used as a means to make files and information easily available to customers. Accessible by using your favorite Internet browser, the web site contains the following information:

- **Product Support** – Data sheets and errata, application notes and sample programs, design resources, user's guides and hardware support documents, latest software releases and archived software
- **General Technical Support** – Frequently Asked Questions (FAQ), technical support requests, online discussion groups, Microchip consultant program member listing
- **Business of Microchip** – Product selector and ordering guides, latest Microchip press releases, listing of seminars and events, listings of Microchip sales offices, distributors and factory representatives

CUSTOMER CHANGE NOTIFICATION SERVICE

Microchip's customer notification service helps keep customers current on Microchip products. Subscribers will receive e-mail notification whenever there are changes, updates, revisions or errata related to a specified product family or development tool of interest.

To register, access the Microchip web site at www.microchip.com. Under "Support", click on "Customer Change Notification" and follow the registration instructions.

CUSTOMER SUPPORT

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Field Application Engineer (FAE)
- Technical Support

Customers should contact their distributor, representative or Field Application Engineer (FAE) for support. Local sales offices are also available to help customers. A listing of sales offices and locations is included in the back of this document.

Technical support is available through the web site at: http://microchip.com/support

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

ISBN: 978-1-5224-4081-9

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
═ ISO/TS 16949 ═

Microchip received ISO/TS-16949:2009 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.

Worldwide Sales and Service

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
http://www.microchip.com/support
Web Address:
www.microchip.com

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Austin, TX
Tel: 512-257-3370

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Dallas
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit
Novi, MI
Tel: 248-848-4000

Houston, TX
Tel: 281-894-5983

Indianapolis
Noblesville, IN
Tel: 317-773-8323
Fax: 317-773-5453
Tel: 317-536-2380

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608
Tel: 951-273-7800

Raleigh, NC
Tel: 919-844-7510

New York, NY
Tel: 631-435-6000

San Jose, CA
Tel: 408-735-9110
Tel: 408-436-4270

Canada - Toronto
Tel: 905-695-1980
Fax: 905-695-2078

ASIA/PACIFIC

Australia - Sydney
Tel: 61-2-9868-6733

China - Beijing
Tel: 86-10-8569-7000

China - Chengdu
Tel: 86-28-8665-5511

China - Chongqing
Tel: 86-23-8980-9588

China - Dongguan
Tel: 86-769-8702-9880

China - Guangzhou
Tel: 86-20-8755-8029

China - Hangzhou
Tel: 86-571-8792-8115

China - Hong Kong SAR
Tel: 852-2943-5100

China - Nanjing
Tel: 86-25-8473-2460

China - Qingdao
Tel: 86-532-8502-7355

China - Shanghai
Tel: 86-21-3326-8000

China - Shenyang
Tel: 86-24-2334-2829

China - Shenzhen
Tel: 86-755-8864-2200

China - Suzhou
Tel: 86-186-6233-1526

China - Wuhan
Tel: 86-27-5980-5300

China - Xian
Tel: 86-29-8833-7252

China - Xiamen
Tel: 86-592-2388138

China - Zhuhai
Tel: 86-756-3210040

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-3090-4444

India - New Delhi
Tel: 91-11-4160-8631

India - Pune
Tel: 91-20-4121-0141

Japan - Osaka
Tel: 81-6-6152-7160

Japan - Tokyo
Tel: 81-3-6880- 3770

Korea - Daegu
Tel: 82-53-744-4301

Korea - Seoul
Tel: 82-2-554-7200

Malaysia - Kuala Lumpur
Tel: 60-3-7651-7906

Malaysia - Penang
Tel: 60-4-227-8870

Philippines - Manila
Tel: 63-2-634-9065

Singapore
Tel: 65-6334-8870

Taiwan - Hsin Chu
Tel: 886-3-577-8366

Taiwan - Kaohsiung
Tel: 886-7-213-7830

Taiwan - Taipei
Tel: 886-2-2508-8600

Thailand - Bangkok
Tel: 66-2-694-1351

Vietnam - Ho Chi Minh
Tel: 84-28-5448-2100

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

Finland - Espoo
Tel: 358-9-4520-820

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Garching
Tel: 49-8931-9700

Germany - Haan
Tel: 49-2129-3766400

Germany - Heilbronn
Tel: 49-7131-67-3636

Germany - Karlsruhe
Tel: 49-721-625370

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Germany - Rosenheim
Tel: 49-8031-354-560

Israel - Ra'anana
Tel: 972-9-744-7705

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Italy - Padova
Tel: 39-049-7625286

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Norway - Trondheim
Tel: 47-7288-4388

Poland - Warsaw
Tel: 48-22-3325737

Romania - Bucharest
Tel: 40-21-407-87-50

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

Sweden - Gothenberg
Tel: 46-31-704-60-40

Sweden - Stockholm
Tel: 46-8-5090-4654

UK - Wokingham
Tel: 44-118-921-5800
Fax: 44-118-921-5820