Application Report
SPRAA90–October 2006

Software Operation of Gigabit Ethernet Media Access Controller on TMS320C645x DSP

Magdalena Iovescu, Mike Denio

ABSTRACT

The TMS645x devices provide an efficient interface between the DSP core processor and the network via a high performance Gigabit Ethernet Media Access Controller (EMAC), supporting four Media Independent Interfaces to the physical layer device (PHY).

This application report discusses the software interface used to operate the EMAC and Management Data Input/Output (MDIO) modules. It describes in detail how to initialize and maintain Ethernet operation in a software application or device driver. Special attention is paid to configuring features new to Gigabit EMAC, and to initializing each of the four Media Independent Interfaces.

There are many different approaches in structuring an Ethernet software application or device driver. This application report is not intended as the only possible methodology; it is rather an example driver, with several example applications that show how to use this driver's APIs, and it is useful to rapidly test and benchmark the EMAC on C645x devices.

This application report and the code that comes with it is not related to the C6000 TCP/IP Stack, although it parallels the HAL driver in the stack software. The example driver described here serves two main purposes:

- Provides a quick hardware check, to verify that the EMAC and its connection to the board and the PHY operates correctly. This is especially useful for customers who spin their own board, and need a way to check the EMAC/PHY hook-up. A customer would use this code to check that that hardware operates properly before attempting to run a full TCP/IP Stack.
- As example/benchmarking code for the Ethernet MAC, is intended to show how to program and benchmark the peripheral. Customers that do not want to use a full stack, but just need to send packets with a layer 2 driver, or are implementing their own TCP/IP stack driver, and need to know how to program C645x EMAC, would benefit from having an easily available, complete Ethernet driver. The code is also useful to benchmark the peripheral by itself, without the overhead of a TCP/IP Stack.
- Further, this driver shows how to implement features of the EMAC that are not supported in the C6000 TCP/IP Stack HAL driver, like using more than one transmit channel.

This application report contains project code that can be downloaded from http://www.ti.com/lit/zip/SPRAA90.

Contents

List of Figures

List of Tables

Trademarks

Code Composer Studio, DSP/BIOS are trademarks of Texas Instruments.

1 Module Function Overview

The Ethernet Media Access Controller on C645x contains three main modules: EMAC Control Module, MDIO module, and EMAC module. This section summarizes the function of each module.

1.1 EMAC Control Module

The EMAC control module is used for global interrupt enable, and to pace back to back interrupts using an interrupt retrigger count based on the peripheral clock (CPUclock/6). There is also an 8K block of RAM local to the EMAC that is used to hold packet buffer descriptors.

Although the EMAC control module and the EMAC module have slightly different functions, they are not distinguished from each other in the driver API. Also, in practice, the type of maintenance performed on the EMAC control module is more commonly conducted from the EMAC module software (as opposed to the MDIO module).

1.2 MDIO Module

The MDIO module is used to initially configure the external PHY device, monitor the PHY, and relay any changes back to the software controlling the EMAC module. The MDIO module software can be a simple implementation to maintain one specific PHY or can maintain the status of multiple PHYs and auto-select the best PHY for use at any given time.

1.3 EMAC Module

The EMAC module is used to send and receive Ethernet packets. This is done by maintaining up to 8 transmit and receive descriptor queues. The EMAC module configuration must also be kept up–to–date based on PHY negotiation results returned from the MDIO module.

2 Software Directory Structure

The EMAC software presented in this document consists of the EMAC/MDIO peripheral driver, and several example applications, highlighting various ways to configure and use the driver API. The directory structure for the code is shown in the Figure 1.

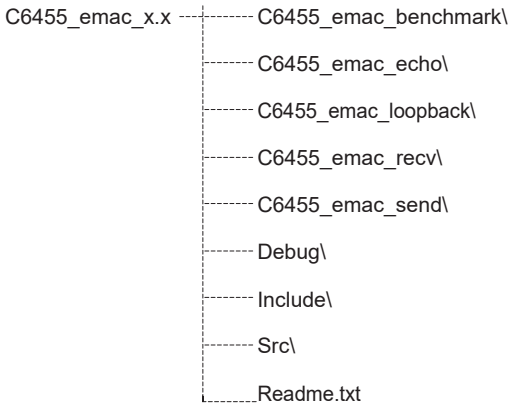

Figure 1. EMAC Software Directory Structure

`C6455_emac_benchmark` directory contains an example application that is used to obtain performance data for both the 10/100 Mbps and Gigabit interfaces of the EMAC. For details on the performance of EMAC when using each of the four interfaces, see Section 8.

`C6455_emac_echo` directory shows an example of how the EMAC on C645x can communicate with another host, in this case a PC. In this application, the DSP sends packets out, which are echoed back by a PC connected to the same subnet. The PC runs the `Udpflood.exe` application. For instructions on how to set up the PC, and run this example, see Section 7.2.

`C6455_emac_loopback` contains the simplest of the example applications, and transfers packet data in loopback mode, using either a loopback cable plugged into the RJ-45 connector, the internal loopback at the PHY level, or a loopback internal to the EMAC peripheral. This application is useful in debugging any hardware issues. This example is described more in Section 7.1.

`C6455_emac_recv` and `C6455_emac_send` examples show how to communicate between two DSPs using the EMAC peripheral.

`Debug` directory is used to store temporary object files. The executable .OUT files are also placed here, for each of the example projects.

`Include` directory contains the header files for the EMAC low-level driver.

`Src` directory contains the source C files for the driver.

3 Target Environment

For the purposes of the example code in this document, some assumptions are made about the target environment. These assumptions are based on the most commonly used configuration of the device (and actually go beyond the base functionality of a device driver). The desired feature set of the target environment is listed below. This is not intended to represent all the potential features of the EMAC system, but only those most commonly used in an application.

- The EMAC module uses a DSP interrupt for servicing transmit, receive, and EMAC status events.
- The MDIO module uses a *100 ms* polling loop to update PHY selection and status monitoring.
- There can be one or more PHYs connected to the DSP (although only one is in use at any given time).
- There is a single receive channel for unicast, broadcast, multicast, and promiscuous packets.
- The driver will not receive any type of error packets.
- There are eight transmit channels. These can be placed in round–robin or fixed–priority mode. The mode is selected at run time.

4 EMAC Control Module Operation

The EMAC control module is used to control device interrupts. The EMAC control module registers are considered part of the EMAC module, and its initialization is combined with that of the EMAC module. For the EMAC driver presented in this document. the code to initialize the EMAC and EMAC control modules is included in **src\c6455_emac.c** file, in the `EMAC_open()` function.

4.1 Initialization

The initialization of the EMAC control module consists of two parts:

- Configuration of the interrupt on the DSP.
- Initialization of the EMAC control module:
  - Setting the interrupt pace count (using EWINTTCNT)
  - Initializing the EMAC and MDIO modules.
  - Enabling interrupts in the EMAC control module (using EWCTL)

The code to perform these actions may appear as in Example 1.

The process of mapping the EMAC interrupts to one of the DSP's interrupts is done using the system's interrupt controller. Once the interrupt is mapped to a DSP interrupt, general masking and unmasking of the interrupt (to control reentrancy) is done at the DSP level, by manipulating the DSP interrupt enable mask. In the example applications, BIOS APIs are used to protect reentrant code, and to enable/disable EMAC interrupts.

The EMAC control module control register (EWCTL) is only used to enable and disable interrupts from within the EMAC interrupt service routine (ISR). This is because disabling and re-enabling the interrupt in EWCTL also resets the interrupt pace counter.

Example 1. EMAC Control Module Initialization Code

```
/*
// Globally disable EMAC/MDIO interrupts in the control module
*/
CSL_FINST( ECTL_REGS->EWCTL, ECTL_EWCTL_INTEN, DISABLE ) ;

/* Reset the EMAC */
EMAC_REGS->SOFTRESET = 0x00000001;
while (EMAC_REGS->SOFTRESET != 0x00000000) ;

/* Set Interrupt Timer Count (CPUclk/6) */
ECTL_REGS->EWINTTCNT = 1500 ;

/*
// Initialize MDIO and EMAC Module
*/

[Discussed Later in this document]

/* Enable global interrupt in the control module */
CSL_FINST(ECTL_REGS->EWCTL, ECTL_EWCTL_INTEN, ENABLE ) ;
```

4.2 Monitoring

There is little monitoring that needs to be done on the EMAC control module. The EMAC driver uses the EMAC control module internal RAM for its packet buffer descriptors, and uses EWCTL and EMAC control module interrupt timer count register (EWINTTCNT) to control interrupts and interrupt pacing from within the EMAC ISR. For the EMAC driver discussed here, the timer count register is not set in the driver code, but rather in the example applications, as the requirements for the interrupt pacing may be different for different applications.

5 MDIO Module Operation

The MDIO module is used to configure and monitor one or more PHY devices that are connected to the EMAC module.

The MDIO software described in this document is written to be a stand–alone module that acts as a slave to the EMAC software. After being initially configured, the MDIO software is entirely autonomous. Changes in PHYs or PHY link state are communicated back to the EMAC module as a return value from the MDIO event processor. The EMAC module can then retrieve the current MDIO state by calling a status function.

This section is not intended to be a primer on PHYs nor PHY control registers. It is intended to document the operation of the MDIO hardware module. It is assumed you have some knowledge of PHY operation. See your PHY device documentation for more information on PHY control registers.

5.1 Initialization

Other than initializing the software state machine (that is beyond the scope of this document), all that needs to be done for the MDIO module is to enable the MDIO engine and to configure the clock divider. To set the clock divider, supply an MDIO clock of 1 MHz. Since the base clock used is the peripheral clock (CPUclk/6), the divider can be set to 166 for a 1 GHz device, with slower MDIO clocks for slower CPU frequencies being perfectly acceptable.

Both the state machine enable and the MDIO clock divider are controlled through the MDIO control register (CONTROL). If none of the potentially connected PHYs require the access preamble, the PREAMBLE bit can also be set in CONTROL to speed up PHY register access. The code for this is included in the `\src\c6455_mdio.c` file, under MDIO_open() function, and may appear as in Example 2.

Example 2. MDIO Module Initialization Code

```
#define VBUSCLK 165

...

/* Enable MDIO and setup divider */
MDIO_REGS->CONTROL =    CSL_FMKT( MDIO_CONTROL_ENABLE, YES) |
                      CSL_FMK( MDIO_CONTROL_CLKDIV, VBUSCLK ) ;
```

If the MDIO module is to operate on an interrupt basis, the interrupts can be enabled at this time using the USERINTMASKSET register for register access and the USERPHYSEL*n* register if a target PHY is already known.

However, to run the software state machine, a real–time–based timer event is required. For this example, the entire MDIO software engine is powered off a 0.1–second timer. Also, the software auto-selects a PHY to use so that the PHY address on the MDIO bus does not have to be specified at run time.

5.2 Selecting and Configuring a PHY

Once the MDIO state machine has been enabled, it starts polling all 32 PHY addresses on the MDIO bus, looking for active PHYs. Since it can take up to 50 μs to read one register, it can be some time before the MDIO state machine provides an accurate representation of all the PHYs available. Also, a PHY can take up to 3 seconds to negotiate a link. Thus, it is advisable to run the MDIO software off a time–based event rather than polling.

5.2.1 PHY Search

The code in Example 3, taken from `MDIO_timerTick()` function, is run when the software state machine is in its initialization state. It reads the MDIO PHY alive indication register (ALIVE) to get a representation of the PHYs that are currently present on the MDIO bus. Over time, the value of this register can change. Thus, the software must re-read the ALIVE register whenever it needs to find a new PHY.

If the corresponding bit is set in the ALIVE register, this code attempts to initialize the PHY based on the input configuration. If the configuration was successful, the PHY search halts while the software state machine waits for a link indication on the target PHY.

Example 3. PHY Search Code

```
// Try the next PHY if anything but a MDIOINIT condition

ltmp1 = MDIO_REGS->ALIVE ;

for( tmp1=0; tmp1<32; tmp1++ )
{
    if( ltmp1 & (1<<pd->phyAddr) )
    {
        if( MDIO_initPHY( pd, pd->phyAddr ) )
              break;
    }

    if( ++pd->phyAddr == 32 )
        pd->phyAddr = 0 ;
}
```

5.2.2 Initial PHY Configuration

The code in Example 3 calls a software function named `MDIO_initPHY()`. This function initializes the PHY and the software state machine. An edited portion of the code is shown in Example 4. The basic process in PHY initial configuration is:

1. Write to the control register of all other active PHY devices (determined by reading the ALIVE register) to isolate them from the MII bus. Although multiple PHYs can share the MDIO bus, they can not share the MII bus.
2. Write to the control register of the target PHY to reset. Wait and verify that the reset completes. This verifies that the PHY is truly alive.
3. Configure other PHY settings, as needed.
4. Read the PHY's capabilities from the PHY status register. Select auto-negotiation or a fix PHY configuration based on the PHY's ability and your preference. This step is done in the `MDIO_initContinue()` function.
5. Begin waiting for negotiation to complete, or for a link condition.

Example 4. PHY Initial Configuration Code

```
/* Shutdown all other PHYs */
ltmp1 = MDIO_REGS->ALIVE ;

for( i=0; ltmp1; i++,ltmp1>>=1 )
{
    if( (ltmp1 & 1) && (i != phyAddr) )
    {
        PHYREG_write( PHYREG_CONTROL, i, PHYREG_CONTROL_ISOLATE |
                                         PHYREG_CONTROL_POWERDOWN ) ;
        PHYREG_wait() ;
    }
}

/* Reset the PHY we plan to use */
PHYREG_write( PHYREG_CONTROL, phyAddr, PHYREG_CONTROL_RESET ) ;
PHYREG_wait( ) ;

...

/* Read the STATUS register to check auto-negotiation capability */
PHYREG_read( PHYREG_STATUS, phyAddr ) ;
PHYREG_waitResults( tmp1 ) ;

/* For Gigabit PHYs interfaces, read the Extended status register as well */
if ( (macsel == CSL_DEV_DEVSTAT_MACSEL_GMII) ||
     (macsel == CSL_DEV_DEVSTAT_MACSEL_RGMII) )
{
    PHYREG_read( PHYREG_EXTSTATUS, pd->phyAddr ) ;
    PHYREG_waitResults( tmp1gig ) ;
}

/* See if we auto-negotiate or not */
if( (pd->ModeFlags & MDIO_MODEFLG_AUTONEG) &&
                    (tmp1 & PHYREG_STATUS_AUTOCAPABLE) )
{
    /* We will use NWAY */
    /* We then "wait" for negotiation to complete */
}
else
{
    /* We will use a fixed configuration */
    /* We then "wait" for a link indication */
}

/* Return success */
return (1) ;
```

5.3 Negotiation Results and Link Indication

Once a PHY has been configured and is either awaiting negotiation or link status, the same state machine checks the status at any given point. The negotiation wait state simply waits for the PHY negotiation to complete. Once this is done, the results of the negotiation are saved and the software state machine enters the link wait state.

The link wait software state just waits for a good link indication from the PHY. This is done by reading the PHY control register. Note that at all times, the MDIO hardware is polling the link state of all PHY devices. The current link state is stored in the MDIO PHY link status register (LINK). The software process for establishing links is:

1. Verify a good link by both reading the PHY status register and by examining the LINK register.
2. Setup to monitor the target PHY using the USERPHYSEL*n* register. This enables tracking of any link state changes using the LINKINTRAW register. Even when polling, it is not possible to miss a link change event.
3. Clear any previously pending LINKINTRAW bit. There can be no race condition, since link would have to go down and come back up between these two operations. Since it takes thousands of CPU cycles to read the PHY, it can not happen.
4. Begin periodic polling of the LINKINTRAW and LINK registers to look for further link changes. There is no need to access the PHY directly from this point forward.
5. On a timeout, begin using the ALIVE register to select a PHY candidate.

The code in Example 5 performs this operation using USERPHYSEL0.

Example 5. Link Indication Code

```
/* Read the STATUS register to check for "link" */
PHYREG_read( PHYREG_STATUS, pd->phyAddr );
PHYREG_waitResultsAck( tmp1, ack ) ;

if( !(tmp1 & PHYREG_STATUS_LINKSTATUS) )
    goto CheckTimeout ;

/* Make sure we are linked in the MDIO module as well */
ltmp1 = MDIO_REGS->LINK ;
if( !( ltmp1&(1<<pd->phyAddr ) ) )
    goto CheckTimeout ;

/* Start monitoring this PHY */
MDIO_REGS->USERPHYSEL0 = pd->phyAddr ;

/* Clear the link change flag so we can detect a "re-link" later */
MDIO_REGS->LINKINTRAW = 1 ;
```

5.4 Monitoring (Event Processing)

The MDIO software module from which the code examples are taken is written such that a central event function handles all parts of the PHY operation. This event function is called every 0.1 second. When in the “linked” software state, the only operation to be performed is to check the status of the LINKINTRAW register for link status changes. When the LINKINTRAW register indicates a change of status or the LINK register indicates no current link, the following operations are performed:

1. If using auto-negotiation and the link is currently down, then restart negotiation; otherwise, re-read negotiation results.
2. Wait for negotiation results when appropriate, or just wait for link.
3. (Execute the same code as in Section 5.3).

The code in Example 6, taken from `MDIO_timerTick( )` function, performs this operation. Most of the actions taken on a link change event are executed by code from Section 5.3.

Example 6. Link Status Monitoring Code

```
/*
// Here we check for a "link-change" status indication or a link
// down indication.
*/
ltmp1 = MDIO_REGS->LINKINTRAW & 1 ;
MDIO_REGS->LINKINTRAW = ltmp1 ;

if( ltmp1 || !(MDIO_RGET(LINK)&(1<<pd->phyAddr)) )
{
/*
// There has been a change in link (or it is down)
// If we do not auto-negotiation, then we just wait for a new link.
// Otherwise, we enter NWAYSTART or NWAYWAIT
*/

/* If not auto-negotiation, just wait for link */
if( !(pd->ModeFlags & MDIO_MODEFLG_NWAYACTIVE) )
      pd->phyState = PHYSTATE_LINKWAIT ;
else
{
      /* Handle auto-negotiation condition */

      /* First see if link is really down */

      PHYREG_read( PHYREG_STATUS, pd->phyAddr ) ;
      PHYREG_waitResults( tmp1 ) ;

      if( !(tmp1 & PHYREG_STATUS_LINKSTATUS) )
      {
      /* No Link - restart auto-negotiation */
      pd->phyState = PHYSTATE_NWAYSTART ;

      PHYREG_write( PHYREG_CONTROL, pd->phyAddr,
                    PHYREG_CONTROL_AUTONEGEN
                    PHYREG_CONTROL_AUTORESTART ) ;
                    PHYREG_wait() ;
      }
      else
      {
      /* We have a Link - re-read auto-negotiation parameters */
      pd->phyState = PHYSTATE_NWAYWAIT ;
      }
 }
}
```

5.5 MDIO Register Access

All of the routines previously described use the MDIO module to access PHY control registers. This is done by using the USERACCESS*n* register. The software functions that implement the access process are five macros:

PHYREG_read (regadr, phyadr)	Start the process of reading a PHY register
PHYREG_write(regadr, phyadr, data)	Start the process of writing a PHY register
PHYREG_wait( )	Synchronize operation (make sure read/write is idle)
PHYREG_waitResults(results)	Wait for read to complete and return data read
PHYREG_waitResultsAck(results, ack)	Wait for read to complete, return data read, and acknowledge the transaction

A wait is not necessary after a write operation, as long as the status is checked before every operation to make sure the MDIO hardware is idle. An alternative approach is to call `PHYREG_wait()` after every write, and `PHYREG_waitResults()` after every read; then the hardware can be assumed to be idle when starting a new operation.

The macros are defined in Example 7 (USERACCESS0 is assumed).

The ACK bit is not checked on PHY register reads. Since the ALIVE register is used to initially select a PHY, it is assumed that the PHY is acknowledging read operations. It is possible that a PHY could become inactive at a future point in time. An example of this would be a PHY that can have its MDIO addresses changed while the system is running. It is not very likely, but this condition can be tested by periodically checking the PHY state in the ALIVE register.

Example 7. MDIO Register Access Macros

```
#define PHYREG_read( regadr, phyadr )                                          \
       MDIO_REGS->USERACCESS0 =                                               \
              CSL_FMK(MDIO_USERACCESS0_GO,1u)          |                       \
              CSL_FMK(MDIO_USERACCESS0_REGADR,regadr) |                        \
              CSL_FMK(MDIO_USERACCESS0_PHYADR,phyadr)

#define PHYREG_write( regadr, phyadr, data )                                   \
       MDIO_REGS->USERACCESS0 =                                               \
              CSL_FMK(MDIO_USERACCESS0_GO,1u)          |                       \
              CSL_FMK(MDIO_USERACCESS0_WRITE,1)        |                       \
              CSL_FMK(MDIO_USERACCESS0_REGADR,regadr) |                        \
              CSL_FMK(MDIO_USERACCESS0_PHYADR,phyadr) |                        \
              CSL_FMK(MDIO_USERACCESS0_DATA, data)

#define PHYREG_wait( )                                                         \
       while( CSL_FEXT(MDIO_REGS->USERACCESS0,MDIO_USERACCESS0_GO) )

#define PHYREG_waitResults( results ) {                                        \
       while( CSL_FEXT(MDIO_REGS->USERACCESS0,MDIO_USERACCESS0_GO) ) ;         \
       results = CSL_FEXT(MDIO_REGS->USERACCESS0, MDIO_USERACCESS0_DATA) ;     \
}

#define PHYREG_waitResultsAck( results, ack ) {                                \
       while( CSL_FEXT(MDIO_REGS->USERACCESS0,MDIO_USERACCESS0_GO) ) ;         \
       results = CSL_FEXT( MDIO_REGS->USERACCESS0,MDIO_USERACCESS0_DATA ) ;    \
       ack = CSL_FEXT( MDIO_REGS->USERACCESS0, MDIO_USERACCESS0_ACK) ;         \
}
```

6 EMAC Module Operation

The EMAC module is used to send and receive data packets over the network. Most of the work in developing an application or device driver for Ethernet is programming this module. The software described is written to implement a basic Ethernet driver. The code is straight forward and non–reentrant. It is assumed that all reentrancy exclusion methods are handled external to this module.

6.1 Initialization

The following is the initialization procedure to get the EMAC to the state where it is ready to receive and send Ethernet packets. Some of these steps are not necessary when performed immediately after device reset.

1. If enabled, clear the device interrupt enable in EWCTL register.
2. Clear the MACCONTROL, RXCONTROL, and TXCONTROL registers (*not necessary immediately after reset*).
3. Initialize all 16 header descriptor pointer registers (RX*n*HDP and TX*n*HDP) to 0.
4. Clear all 36 statistics registers by writing 0 *(not necessary immediately after reset)*.
5. Initialize all 32 receive address RAM locations to 0. Set up the addresses to be matched to the eight receive channels and the addresses to be filtered, through programming the MACINDEX, MACADDRHI, and MACADDRLO registers. When using more than one receive channel, start with channel 0 and progress upwards.
6. Initialize the RX*n*FREEBUFFER, RX*n*FLOWTHRESH, and RXFILTERLOWTHRESH registers, if buffer flow control is to be enabled. Program the FIFOCONTROL register if FIFO flow control is desired. Flow control is not used in this example driver.
7. Most device drivers open with no multicast addresses, so clear MACHASH1 and MACHASH2 registers to 0.
8. Write the RXBUFFEROFFSET register value (typically zero).
9. Initially clear all unicast channels by writing FFh to the RXUNICASTCLEAR register. If unicast is desired, it can be enabled now by writing the RXUNICASTSET register. Some drivers will default to unicast on device open while others will not. In this code, the unicast is enabled, if desired, by calling the `EMAC_setReceiveFilter()` function; details on how to use this function to set up a receive filter are explained in Section 6.2.1.
10. If you desire to transfer jumbo frames, set the RXMAXLEN register to the maximum frame length you want to allow to be received. Jumbo frames are defined as those packets that exceed the standard Ethernet MTU, which is 1500 bytes.
11. Setup the RXMBPENABLE register with an initial configuration. The configuration is based on the current receive filter settings of the device driver. Some drivers may enable things like broadcast and multicast packets immediately, while others may not.
12. Set the appropriate configuration bits in the MACCONTROL register (do not set the GMIIEN bit yet).
13. Clear all unused channel interrupt bits by writing RXINMASKCLEAR and TXINTMASKCLEAR.
14. Enable the receive and transmit channel interrupt bits in RXINTMASKSET and TXINTMASKSET for the channels to be used, and enable the HOSTMASK and STATMASK bits using the MACINTMASKSET register.
15. Initialize the receive and transmit descriptor list queues. There is an infinite number of ways this can be done using the 8K descriptor memory block contained in the EMAC control module. One particular method is detailed later in this section.
16. Prepare receive by writing a pointer to the head of the receive buffer descriptor list to RX*n*HDP. In this example we use only RX0HDP.
17. Enable the receive and transmit DMA controllers by setting the RXEN bit in the RXCONTROL register and the TXEN bit in the TXCONTROL register.
18. Set the RXOWNERSHIP and RXOFFLENBLOCK bits of the MACCONTROL register, if the EMAC receive buffer processing optimization is desired. When this optimization is enabled, the DSP will set the OFFSET/LENGTH and OWNERSHIP fields of the receive buffer descriptor only the first time the descriptor is used; it does not need to set them for every received packet.
19. Set the GMIIEN bit in MACCONTROL.
20. Enable the device interrupt in EWCTL.

The code in Example 8 implements the initialization steps. Some simplifications have been made, but the full source code to the Ethernet module is available in the code delivered with this application report, in the `EMAC_open()` function.

Example 8. EMAC Module Initialization Code

```
/*
// (Step 2) Disable receive, transmit, and clear MACCONTROL
*/
CSL_FINST( EMAC_REGS->TXCONTROL, EMAC_TXCONTROL_TXEN, DISABLE ) ;
CSL_FINST( EMAC_REGS->RXCONTROL, EMAC_RXCONTROL_RXEN, DISABLE ) ;
EMAC_REGS->MACCONTROL = 0 ;

/* (Step 3) Must manually initialize HDPs to NULL */
pRegAddr = &EMAC_REGS->TX0HDP ;
for( i=0; i<8; i++ )
     *pRegAddr++ = 0 ;

pRegAddr = &EMAC_REGS->RX0HDP ;
for( i=0; i<8; i++ )
     *pRegAddr++ = 0 ;

/*
// (Step 4) While GMIIEN is clear in MACCONTROL, we can write directly to
// the statistics registers
*/
pRegAddr = &EMAC_REGS->RXGOODFRAMES ;
for( i=0; i<EMAC_NUMSTATS; i++ )
     *pRegAddr++ = 0 ;

/*
// (Step 5) Setup device MAC address
*/
/* Initialize the RAM locations */
for ( i = 0; i < 32; i++ )
{
     EMAC_REGS->MACINDEX = i ;
     EMAC_REGS->MACADDRHI = 0 ;
     EMAC_REGS->MACADDRLO = 0 ;
}

/* Setup device MAC address */
EMAC_REGS->MACINDEX = 0x0 ;

tmpval = 0 ;
for( i=3; i>=0; i-- )
     tmpval = (tmpval<<8) | localDev.Config.MacAddr[i] ;
EMAC_REGS->MACADDRHI = tmpval ;

tmpval = localDev.Config.MacAddr[5];
EMAC_REGS->MACADDRLO = CSL_FMKT( EMAC_MACADDRLO_VALID, VALID )      |
                       CSL_FMKT( EMAC_MACADDRLO_MATCHFILT, MATCH ) |
                       CSL_FMK(  EMAC_MACADDRLO_CHANNEL, 0 )       |
                       (tmpval<<8)                                 |
                       localDev.Config.MacAddr[4] ;


/* (Step 7) Clear multicast hash bits */
EMAC_REGS->MACHASH1 = 0 ;
EMAC_REGS->MACHASH2 = 0 ;

/* (Step 8) For us buffer offset will always be zero */
EMAC_REGS->RXBUFFEROFFSET = 0 ;
```

Example 8. EMAC Module Initialization Code (continued)

```
/* (Step 9) Clear Unicast receive on channel 0-7 */
EMAC_REGS->RXUNICASTCLEAR = 0xFF ;

/* (Step 10) Set the maximum length frames allowed */
#if USE_JUMBO
    EMAC_REGS->RXMAXLEN = 0x7fff ;
#endif

/* (Step 11) Reset receive (M)ulticast (B)roadcast (P)romiscuous Enable register */
EMAC_REGS->RXMBPENABLE = 0 ;

/* Set the pass receive CRC mode and adjust maximum buffer accordingly */
if( localDev.Config.ModeFlags & EMAC_CONFIG_MODEFLG_RXCRC )
{
    CSL_FINST( EMAC_REGS->RXMBPENABLE, EMAC_RXMBPENABLE_RXPASSCRC, INCLUDE ) ;
    localDev.PktMTU = OURMTU+4 ;
}
else
    localDev.PktMTU = OURMTU ;

/* (Step 12) Set the channel configuration to priority if requested */
if( localDev.Config.ModeFlags & EMAC_CONFIG_MODEFLG_CHPRIORITY )
    CSL_FINST( EMAC_REGS->MACCONTROL, EMAC_MACCONTROL_TXPTYPE, CHANNELPRI ) ;

/* Set internal EMAC loopback if requested */
if( localDev.Config.ModeFlags & EMAC_CONFIG_MODEFLG_MACLOOPBACK )
    CSL_FINST( EMAC_REGS->MACCONTROL, EMAC_MACCONTROL_LOOPBACK, ENABLE ) ;

/*
// (Step 13 & 14) Enable transmit and receive channel interrupts (set mask  bits)
// We only ever use one receive channel, but up to 8 transmit channels
// Enable Host interrupts
*/
EMAC_REGS->RXINTMASKCLEAR = 0xFF ;
EMAC_REGS->TXINTMASKCLEAR, 0xFF ;
EMAC_REGS->RXINTMASKSET = 1 ;
for( i=0; i<localDev.Config.TxChannels; i++ )
     EMAC_REGS->TXINTMASKSET = (1<<i);
EMAC_REGS->MACINTMASKSET = CSL_FMK( EMAC_MACINTMASKSET_HOSTMASK, 1 ) |
                           CSL_FMK( EMAC_MACINTMASKSET_STATMASK, 1 ) ;

/*
// (Step 15) Setup Receive Buffers and Transmit Buffers
*/

[Discussed Later in this document]

/* (Step 16) Prepare receive */
EMAC_REGS->RX0HDP = (Uint32)localDev.RxCh.pDescRead ;

/*
// (Step 17) Enable receive, transmit, and GMII
*/
CSL_FINST( EMAC_REGS->TXCONTROL, EMAC_TXCONTROL_TXEN, ENABLE ) ;
CSL_FINST( EMAC_REGS->RXCONTROL, EMAC_RXCONTROL_RXEN, ENABLE ) ;

/*
// (Step 18) Enable receive buffer descriptor optimization
*/
#if USE_GMAC_OPT
    CSL_FINST( EMAC_REGS->MACCONTROL, EMAC_MACCONTROL_RXOWNERSHIP, ONE ) ;
    CSL_FINST( EMAC_REGS->MACCONTROL, EMAC_MACCONTROL_RXOFFLENBLOCK, BLOCK ) ;
#endif
```

Example 8. EMAC Module Initialization Code (continued)

```
/*
// (Step 19) Enable GMII
*/

CSL_FINST( EMAC_REGS->MACCONTROL, EMAC_MACCONTROL_GMIIEN, ENABLE ) ;
```

6.2 Packet Receive Configuration

The example code given in the previous section assumes that the EMAC is being initialized in a (mostly) idle state, and that it can not receive any type of Ethernet packet (unicast, broadcast, or multicast) in its default state. The software interface from which the example code is taken provides two functions to configure packet reception, `setReceiveFilter()` and `setMulticast()`.

6.2.1 Setting the Receive Filter

There are two approaches to a receive filter in an Ethernet device driver. One approach is to treat unicast, broadcast, and multicast packets as all individual entities. The second approach is to treat each receive level as being inclusive of the previous level. This example takes the second approach.

Regardless of the software approach, the RXUNICASTSET, RXUNICASTCLEAR, RXMBPENABLE, and MACHASH*n* registers are used to control unicast, broadcast, multicast, and promiscuous operations.

This code example assumes a filter value set as follows. Each successive filter includes the previous, so the effect is cumulative:

```
#define EMAC_RXFILTER_NOTHING                   0  /* Receive nothing                  */
#define EMAC_RXFILTER_DIRECT                    1  /* Receive unicast packets          */
#define EMAC_RXFILTER_BROADCAST                 2  /* Above plus broadcast packets     */
#define EMAC_RXFILTER_MULTICAST                 3  /* Above plus specified multicast   */
#define EMAC_RXFILTER_ALLMULTICAST              4  /* Above plus all multicast         */
#define EMAC_RXFILTER_ALL                       5  /* Any non-error packet             */
```

The C645x EMAC has two ways of setting a list of multicast addresses: through 64-bit hash tables and by setting up to 32 multicast addresses to be matched or filtered in the receive address RAM.

The code to set the filter setting (stored in the variable `ReceiveFilter`) is shown in Example 9. The logic is to disable anything that is not set, and then enable anything that is set. When receiving a specified list of multicast addresses, the addresses can be either stored in the address RAM, or the bits representing the specified list may be stored in `pd->MacHash1` and `pd->MacHash2`, depending on the multicasting methodology desired. The code to calculate these values for both approaches is discussed in the next section.

Example 9. Setting the Receive Filter Code

```
/*
// The following code relies on the numeric relation of the filter
// value such that the higher filter values receive more types of
// packets.
*/
    /* Disable Section */
    if( ReceiveFilter < EMAC_RXFILTER_ALL )
        CSL_FINST(EMAC_REGS->RXMBPENABLE, EMAC_RXMBPENABLE_RXCAFEN, DISABLE ) ;

#if !RAM_MCAST
    if( ReceiveFilter < EMAC_RXFILTER_ALLMULTICAST )
    {
        EMAC_REGS->MACHASH1 = pd->MacHash1 ;
        EMAC_REGS->MACHASH2 = pd->MacHash2 ;
    }
    if( ReceiveFilter < EMAC_RXFILTER_MULTICAST )
        CSL_FINST(EMAC_REGS->RXMBPENABLE, EMAC_RXMBPENABLE_RXMULTEN, DISABLE ) ;
#endif

    if( ReceiveFilter < EMAC_RXFILTER_BROADCAST )
        CSL_FINST(EMAC_REGS->RXMBPENABLE, EMAC_RXMBPENABLE_RXBROADEN, DISABLE ) ;
    if( ReceiveFilter < EMAC_RXFILTER_DIRECT )
        EMAC_REGS->RXUNICASTCLEAR = 1 ;

    /* Enable Section */
    if( ReceiveFilter >= EMAC_RXFILTER_DIRECT )
        EMAC_REGS->RXUNICASTSET = 1 ;
    if( ReceiveFilter >= EMAC_RXFILTER_BROADCAST )
        CSL_FINST(EMAC_REGS->RXMBPENABLE, EMAC_RXMBPENABLE_RXBROADEN, ENABLE ) ;

#if !RAM_MCAST
    if( ReceiveFilter >= EMAC_RXFILTER_MULTICAST )
        CSL_FINST(EMAC_REGS->RXMBPENABLE, EMAC_RXMBPENABLE_RXMULTEN, ENABLE ) ;
#endif

    if( ReceiveFilter >= EMAC_RXFILTER_ALLMULTICAST )
    {
        EMAC_REGS->MACHASH1 = 0xffffffff ;
        EMAC_REGS->MACHASH1 = 0xffffffff ;
    }
    if( ReceiveFilter == EMAC_RXFILTER_ALL )
        CSL_FINST(EMAC_REGS->RXMBPENABLE, EMAC_RXMBPENABLE_RXCAFEN, ENABLE ) ;

    pd->RxFilter = ReceiveFilter ;
```

6.2.2 Setting the Multicast List

The type of multicast desired can be controlled with the RAM_MCAST constant defined in `\src\c6455_emac.c` file. The RAM can be setup to receive up to 32 addresses, which can be a combination of unicast, multicast, or broadcast. For this example, the address at index zero in the RAM was reserved for unicast traffic, and the rest, of up to index 31, can be setup to receive traffic from specific multicast addresses.

When the receive filter is set to EMAC_RXFILTER_ALLMULTICAST, then the hash tables are used to allow for all multicast traffic to be received.

6.2.2.1 Setting the Multicast List Via Hash Tables

To use the hash tables to setup the list of specific multicast addresses, change the following:

```
#define RAM_MCAST    1

to

#define RAM_MCAST    0
```

Sometimes in a device driver, adding and removing addresses from a multicast list can be a single entry at a time. In other device drivers or mini–drivers, the multicast list is maintained by a parent driver or the application and always passed down as a list, as is the case in this example.

The code in Example 10 has a very specific function. It take a list of Ethernet MAC addresses and it hashes each address to calculate a bit to set in the MACHASH*n* register, to allow the EMAC to receive packets destined for that address. The accumulated set of bits to set in MACHASH0 and MACHASH1 are stored in the variables `pd->MacHash1` and `pd->MacHash2` for use in the `setReceiveFilter()` function.

In Example 10, `AddrCnt` is the number of 6–byte MAC addresses in the address list, and `pMCastList` is a pointer to a Uint8, pointing to a concatenated list of MAC addresses (each being 6 bytes in length).

Example 10. Setting the Multicast List With Hash Tables Code

```
Uint8 HashVal,tmpval;

#if !RAM_MCAST

    /* Clear the hash bits */
    pd->MacHash1 = 0 ;
    pd->MacHash2 = 0 ;

    /* For each address in the list, hash and set the bit */
    for( tmp1=0; tmp1<AddrCnt; tmp1++ )
    {
        HashVal=0 ;
        for( tmp2=0; tmp2<2; tmp2++ )
        {
            tmpval = *pMCastList++ ;
            HashVal ^= (tmpval>>2)^(tmpval<<4) ;
            tmpval = *pMCastList++ ;
              HashVal ^= (tmpval>>4)^(tmpval<<2) ;
          tmpval = *pMCastList++ ;
            HashVal ^= (tmpval>>6)^(tmpval) ;
        }
        if( HashVal & 0x20 )
            pd->MacHash2 |= (1<<(HashVal&0x1f)) ;
        else
            pd->MacHash1 |= (1<<(HashVal&0x1f)) ;
    }

    /* We only write the hash table if the filter setting allows */
    if( pd->RxFilter < EMAC_RXFILTER_ALLMULTICAST )
    {
        EMAC_REGS->MACHASH1 = pd->MacHash1 ;
        EMAC_REGS->MACHASH2 = pd->MacHash2 ;
    }
#endif
```

6.2.2.2 Setting the Multicast List via RAM Address

When RAM_MCAST is defined (which is the default), up to 31 multicast addresses can be set in the address RAM.

The code in Example 11 shows how to setup the multicast traffic to be matched on the receive channel 0.

Example 11. Setting the Multicast List in RAM Code

```
Uint32 temp,temp1;

#if RAM_MCAST

if (AddrCnt > 31)
   return ( EMAC_ERROR_INVALID ) ;

/* Clear the multicast list */
for (i = 1; i < 32; i++)
{
     EMAC_REGS->MACINDEX = i ;
     EMAC_REGS->MACADDRHI = 0 ;
     EMAC_REGS->MACADDRLO = 0 ;
}

/* For each address in the list, add it to the RAM */
for( tmp1=0; tmp1<AddrCnt; tmp1++ )
{

    EMAC_REGS->MACINDEX = tmp1+1 ;
    temp = 0;

    for( i=3; i>=0; i- - )
    temp = (temp<<8) | *(pMCastList+i) ;

    EMAC_REGS->MACADDRHI = temp ;

    temp = *(pMCastList+4) ;
    temp1 = *(pMCastList+5) ;
    EMAC_REGS->MACADDRLO=CSL_FMKT (EMAC_MACADDRLO_VALID, VALID)      |
                         CSL_FMKT (EMAC_MACADDRLO_MATCHFILT, MATCH) |
                         CSL_FMK (EMAC_MACADDRLO_CHANNEL, 0)        |
                         (temp1<<8) | temp;pMCastList+=6;}# end if
    pMCastList+=6

    }

#endif
```

6.3 Receive

The reception of Ethernet packets is performed through the use of a buffer descriptor system where the application software or device driver describes empty memory buffers to the EMAC to which Ethernet packet data can be written. The buffer descriptor is a 16–byte memory structure that is stored in a 8K–byte memory space contained in the EMAC control module. The EMAC control module has space for up to 512 descriptors. You should be familiar with the EMAC operational overview and the detailed description of the receive buffer descriptor fields, which can be found in the *TMS320C645x DSP Ethernet Media Access Controller (EMAC) / Management Data Input/Output (MDIO) User's Guide* (SPRU975).

There are a number of ways in which the descriptor memory contained in the EMAC control module can be managed. One option would be to write a memory allocation system where 16–byte descriptors are allocated and freed as needed, so that a descriptor may be used for a receive buffer at one point, and then a totally different transmit buffer the next. Another option would be to statically allocate packet buffers and permanently assign a descriptor slot to each. This way, the descriptor's pointer to the packet buffer would never have to be updated.

The method used in this example code uses a third option. Here, the 512 descriptor slots available in the control module are divided in an arbitrary method where each receive or transmit channel has its own set of descriptors. The descriptor structure is:

```
/*
// Transmit/Receive Descriptor Channel Structure
*/
typedef struct _EMAC_DescCh {
    struct _EMAC_Device *pd ;          /* Pointer to parent structure               */
    PKTQ                DescQueue ;    /* Packets queued as descriptors             */
    PKTQ                WaitQueue ;  /* Packets waiting for transmit descriptors   */
    uint                ChannelIndex ; /* Channel index 0-7                         */
    uint                DescMax ;       /* Max number of descriptors (buffers)      */
    uint                DescCount ;    /* Current number of descriptors             */
    EMAC_Desc           *pDescFirst ;  /* First descriptor location                 */
    EMAC_Desc           *pDescLast ;   /* Last descriptor location                  */
    EMAC_Desc           *pDescRead ;   /* Location to read next descriptor          */
    EMAC_Desc           *pDescWrite ;  /* Location to write next descriptor         */
} EMAC_DescCh ;
```

For a receive channel, each descriptor refers to a fixed length buffer that is always at least OURMTU or OURMTU+4 bytes in length (depending on whether CRC is included in the data or not). OURMTU is 1514 for a standard Ethernet packet, and can be up to 30K for jumbo packets. The size of the jumbo packets allowed is further limited by the PHY capabilities. For example, the Broadcom PHY used on the C6455 EVM board allows for packets up to 10K.

Therefore, each descriptor represents one packet. There is a fixed number of descriptors and that number represents the maximum number of packets that can be received before the receive interrupt needs to be serviced.

Figure 2 illustrates some of the descriptor fields. Each receive channel has a fixed number of slots. When a packet is received and handed over to the software for processing, a fresh empty buffer is pulled from a central pool, and the descriptor slot is reused to point to the new buffer. The `DescQueue` field in the structure is a queue of physical packet buffers (in DSP memory) that are currently being "described" by the descriptor list. Since there is no queue of free buffers for any given receive channel (other than those already contained in the descriptor list), the `WaitQueue` is not used.

The descriptors are tracked via the variables, `pDestFirst`, pDestLast, `pDescRead`, and `pDescWrite`. The `pDescFirst` and `pDescLast` pointers just point to the first and last descriptors in the fixed circular queue. These never change. The `pDescRead` pointer points to the next descriptor buffer that may contain a new packet received from the network. The `pDescWrite` pointer points to the descriptor to use when adding the next empty buffer to the queue.

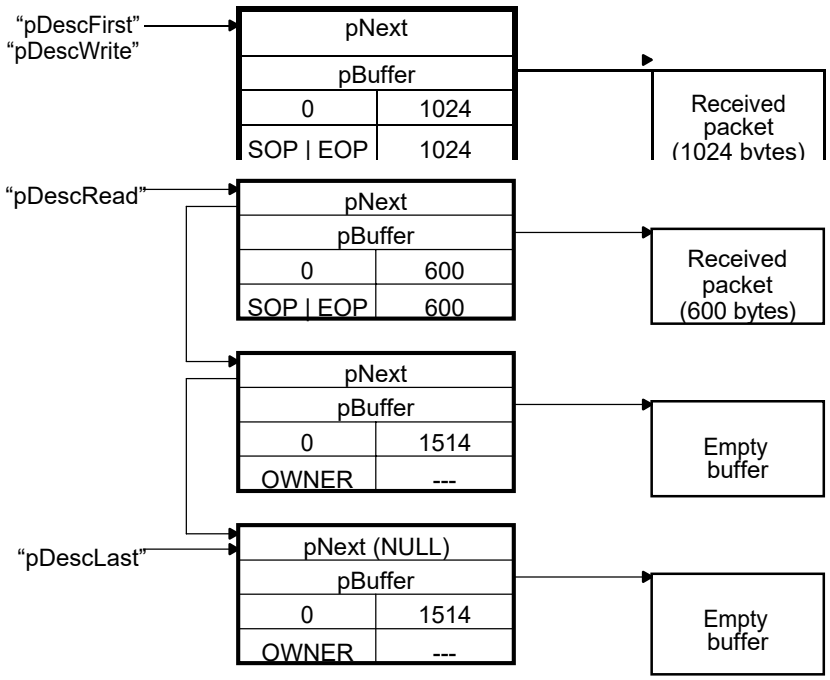

Figure 2. Receive Descriptor Linked List

In Figure 2, there are four descriptors allocated to the receive channel. Of these, only three descriptors are in use. One of the descriptors has already received a packet that has been handed up to the software. This descriptor is currently not in use. The next descriptor has received a packet, but has not been serviced yet by the software. The final two descriptors point to empty data buffers and are waiting to receive packet data from the EMAC.

In practice, the software always tries to keep all descriptors pointing to empty buffers. This allows the EMAC to run longer without being serviced and without experiencing a packet overrun condition.

For servicing a receive channel, there are two basic functions. The first function is called `emacEnqueueRx`(). Its job is to fill all possible receive descriptor slots so that they point to empty packet buffers. The second function is called `emacDequeueRx`(). Its job is to pull buffers from the list that have received packet data, and to update the corresponding descriptor so that is points to a new empty buffer.

It is helpful to consider how packet buffers are represented in the code. The example code in Example 12 uses a structure of type `EMAC_Pkt` to define a packet. This structure has little to do with the EMAC hardware, but must be understood to follow the software examples. The structure and its related flags are defined below. Note that it is significantly similar to the descriptor format.

Example 12. Receive Packets Example Code

```
typedefstruct_EMAC_Pkt{
     struct_EMAC_Pkt *pPrev;                /*Previous record                          */
     struct_EMAC_Pkt *pNext;                /*Next record                              */
     Uint8           *pDataBuffer;          /*Pointer to Data Buffer                   */
     Uint32          BufferLen;             /*Physical length of buffer (read only)    */
     Uint32          Flags;                 /*Packet flags                             */
     Uint32          ValidLen;              /*Length of valid data in buffer           */
     Uint32          DataOffset;            /*Byte off set to valid data               */
     Uint32          PktChannel;            /*Transmit channel/Priority 0-7 (SOPonly)  */
     Uint32          PktLength;             /*Length of packet (SOPonly)               */
     Uint32          PktFrags;              /*Number of fragments in packet (SOPonly)  */
}EMAC_Pkt;

/*
//Packet Buffer Flags set in Flags
*/
#define EMAC_PKT_FLAGS_SOP                  0x80000000u /*  Startofpacket              */
#define EMAC_PKT_FLAGS_EOP                  0x40000000u /*  Endofpacket                */

/*
//The following packet flags are set in Flags on receive packets only
*/
#define EMAC_PKT_FLAGS_HASCRC               0x04000000u /*  RxCrc: PKT has 4 byte CRC  */
#define EMAC_PKT_FLAGS_JABBER               0x02000000u /*  RxErr:Jabber               */
#define EMAC_PKT_FLAGS_OVERSIZE             0x01000000u /*  RxErr:Oversize             */
#define EMAC_PKT_FLAGS_FRAGMENT             0x00800000u /*  RxErr:Fragment             */
#define EMAC_PKT_FLAGS_UNDERSIZED           0x00400000u /*  RxErr:Undersized           */
#define EMAC_PKT_FLAGS_CONTROL              0x00200000u /*  RxCtl:ControlFrame         */
#define EMAC_PKT_FLAGS_OVERRUN              0x00100000u /*  RxErr:Overrun              */
#define EMAC_PKT_FLAGS_CODEERROR            0x00080000u /*  RxErr:CodeError            */
#define EMAC_PKT_FLAGS_ALIGNERROR           0x00040000u /*  RxErr:AlignmentError       */
#define EMAC_PKT_FLAGS_CRCERROR             0x00020000u /*  RxErr:BadCRC               */
#define EMAC_PKT_FLAGS_NOMATCH              0x00010000u /*  RxPrm:NoMatch              */
```

6.3.1 Enqueue Receive Descriptor Function

In an ideal system, the only call to an `emacEnqueueRx`() function would occur during initialization. This is because part of the `emacDequeueRx`() function is to keep the descriptor list full of pointers to empty buffers. However, at any given time, an empty buffer may not be available, so one or more descriptor slots allocated to a receive channel can become empty. This was shown in Figure 2 that had one empty descriptor.

To fully understand the enqueue function, Example 13 shows the code from the initialization function that allocates descriptor slots to the one receive channel and multiple transmit channels in the driver environment. Also shown is the call for `emacEnqueueRx`() to fill the descriptors with pointers to empty buffers.

In this code, the variable `localDev.RxCh` is a structure of type `EMAC_DescCh` described earlier.

Example 13. Initialization Code That Allocates Descriptor Slots

```
/*
//Setup Receive Buffers
*/

/*Pointer to first descriptor to use on receive */
pDesc = (EMAC_Desc*)_EMAC_DSC_BASE_ADDR ;

/*Number of descriptors for receive channel */
utemp1 = localDev.Config.RxMaxPktPool ;
```

Example 13. Initialization Code That Allocates Descriptor Slots (continued)

```
/* Init receive */
localDev.RxCh.pd               = &localDev;
localDev.RxCh.DescMax          = utemp1;
localDev.RxCh.pDescFirst       = pDesc;
localDev.RxCh.pDescLast        = pDesc+(utemp1-1);
localDev.RxCh.pDescRead        = pDesc;localDev.RxCh.pDescWrite=pDesc;

/* Fill the descriptor table */
emacEnqueueRx( &localDev.RxCh,0 )
```

The second calling parameter to `emacEnqueueRx`, is a flag indicating that the function is being called at initialization time, and it should not restart the receiver. The only other time the function can be called is from a 0.1 second polling loop (the same that drives the MDIO software state machine). This is done so that if a buffer shortfall occurs, the system looks for new buffers every 0.1 seconds.

Now here is the enqueue function. The process for enqueuing a packet buffer to the descriptor ring is:

1. If the descriptor set is not full, call an application callback to get a free packet buffer.
2. If the packet buffer was obtained, get a pointer to the descriptor to fill from `pDescWrite` and advance the `pDescWrite` pointer while bumping the `DescCount`.
3. Fill in the descriptor with the pointer to the packet buffer. The size is fixed at maximum packet size (`OURMTU` or `OURMTU+4` bytes). When the receive optimization is not used, also set the OWNER flag in the descriptor so that the EMAC knows it can use it. When the optimization is used, the OWNER flag needs to be set only when the receiver is restarted. For more on the receive optimization, see Section 6.1.
4. Make the `pNext` pointer for the new descriptor NULL because it is always the end of the list. Make the previous descriptor in the set point to the new descriptor.
5. Push a structure pointer (handle) to the packet buffer (the thing the descriptor points to) onto its own software queue. The software queue of packet buffer handles is kept synchronized with the list of buffer descriptors. Thus the packet buffer handle can be given back to the application once a packet has been received into the buffer.
6. Return to step 1 until full or there are no more free buffers.
7. As a final step (if not called during initialization): if, when the function was called, all the receive descriptors were used, then the receive engine must be stopped. If new descriptors have been added, then restart the receive engine by posting the head of the descriptor list (`pDescRead`) to RX0HDP.

The source code to implement this function is shown in Example 14.

Example 14. Enque Receive Descriptor Function Code

```
static void emacEnqueueRx ( EMAC_DescCh *pdc, uint fRestart )
{
     EMAC_Pkt     *pPkt ;
     EMAC_Desc    *pDesc ;
     uint         CountOrg ;

     /* Keep the old count around */
     CountOrg = pdc->DescCount ;

     /* Fill receive packets until full */
     while ( pdc->DescCount < pdc->DescMax )
     {

          /* Get a buffer from the application */
          pPkt = (*localDev.Config.pfcbGetPacket) (pdc->pd->hApplication) ;

          /*If no more buffers are available, break out of loop */
          if ( !pPkt ) break ;

          /* Fill in the descriptor for this buffer */
          pDesc = pdc->pDescWrite ;

         /* Move the write pointer and bump count */
         if ( pdc->pDescWrite == pdc->pDescLast )
              pdc->pDescWrite = pdc->pDescFirst ;
         else
              pdc->pDescWrite++ ;
         pdc->DescCount++ ;

         /* Supply buffer pointer with application supplied offset */
         pDesc->pNext        =0 ;
         pDesc->pBuffer      =pPkt->pDataBuffer + pPkt->DataOffset ;
#if USE_EMAC_OPT
         if( !fRestart)
         {
             pDesc->BufOffLen = localDev.PktMTU;
             pDesc->PktFlgLen = EMAC_DSC_FLAG_OWNER;
         }

#else
         pDesc->BufOffLen = localDev.PktMTU ;
         pDesc->PktFlgLen = EMAC_DSC_FLAG_OWNER ;
#endif
         /* Make the previous buffer point to us */
         if( pDesc == pdc->pDescFirst )
              pdc->pDescLast->pNext = pDesc ;
         else
              (pDesc-1)->pNext = pDesc ;
         /* Push the packet buffer on the local descriptor queue */
         pqPush( &pdc->DescQueue, pPkt) ;
   }

   /* Restart receive if we had ran out of descriptors and got here */
   if( fRestart && !CountOrg && pdc->DescCount )
       EMAC_REGS->RX0HDP=(Uint32)pdc->pDescRead ;
}
```

6.3.2 Dequeue Receive Descriptor Function

The `emacDequeueRx()` function is the more interesting of the two receive descriptor–based functions. Its function is to process new packets as they are received by the EMAC, and keep the receive descriptor set always pointing to fresh empty packet buffers.

To understand this function better, it is important to know what happens during a device interrupt. The ISR code relating to the receive operation is:

```
/* Look for receive interrupt (channel 0) */
if( intflags & CSL_FMK( EMAC_MACINVECTOR_RXPEND, 1<<0 ) )
{
    Desc = EMAC_REGS->RX0CP ;
    EMAC_REGS->RX0CP = Desc ;

    emacDequeueRx( &pd->RxCh, (EMAC_Desc *)Desc ) ;
}
```

First, the RXPEND field in the MACINVECTOR register is examined to determine which receive channels have had new activity. In this code, only receive channel 0 is used. Next, the last descriptor to process can be read from RX0CP. This is also the register we write the value of the last descriptor processed. Since the `emacDequeueRx()` function processes all descriptors up to the one that it is passed, the receive interrupt can be immediately acknowledged by writing the value back to the RX0CP register. Finally, the `emacDequeueRx()` function is called with a pointer to the receive descriptor channel structure and a pointer to the last descriptor to service.

The `emacDequeueRx()` function is shown in Example 15. The functions it needs to perform are:

1 The next descriptor to process is always available at the `pDescRead` pointer. The flags for that descriptor are read. Also, the packet buffer that corresponds to the descriptor is popped of the software queue.

2 The `EMAC_Pkt` structure fields are filled in based on the information in the buffer descriptor. If the driver is configured to receive error packets, then the error bits are potentially set in the flags field as well.

3 A pointer to the completed `EMAC_Pkt` structure is passed to the application via a callback function. This function should return a pointer to an identical structure containing a new empty buffer.

4 If this is the last descriptor to process `(pDescRead == pDescAck)`, then a flag is set to prevent the loop from executing again.

5 The `pDescRead` pointer is incremented and the `DescCount` is decremented.

6 If the application did supply a new empty buffer, the buffer is added to the next available descriptor as read from the `pDescWrite` pointer. Under ideal circumstances, this will be the same descriptor that just contained the received packet. However, if there is a free buffer shortage, the read and write pointers will not be synchronized.

7. Next, the descriptor is initialized to point to the empty packet buffer. This code is very similar to that described in Section 6.3.1.

8 Continue until all the descriptors have been processed up to and including that indicated by the caller (in this case the ISR).

9 As a final step, if the last descriptor processed had the EMAC_DSC_FLAG_EOQ flag set in its flags field, this means that the EMAC interpreted the descriptor as being the last in the descriptor chain (its next pointer was NULL). This should not happen under normal operation, but can occur if the system runs out of receive buffers. Since the receive engine stops on this descriptor, it can only happen on the last descriptor to process. When the bit is set, and there are some free buffer descriptors ready, then restart the receive engine by posting the head of the descriptor list (`pDescRead`) RX0HDP.

The source code to implement this function is in Example 15.

Example 15. Dequeue Receive Descriptor Function Code

```
static void emacDequeueRx( EMAC_DescCh *pdc, EMAC_Desc *pDescAck )
{
    EMAC_Pkt     *pPkt ;
    EMAC_Pkt     *pPktNew ;
    EMAC_Desc    *pDesc ;
    uint         tmp ;
    Uint32       PktFlgLen ;

    /* Pop & free buffers untill the last descriptor */
    for( tmp=1; tmp; )
    {
        /* Get the status of this descriptor */
        PktFlgLen = pdc->pDescRead->PktFlgLen ;

        /* Recover the buffer and free it */
        pPkt = pqPop( &pdc->DescQueue ) ;
        if( pPkt )
        {
            /* Fill in the necessary packet header fields */
            pPkt->Flags = PktFlgLen & 0xFFFF0000 ;
            pPkt->ValidLen = pPkt->PktLength = PktFlgLen & 0xFFFF ;
            pPkt->PktChannel = 0 ;
            pPkt->PktFrags = 1 ;

        /* Pass the packet to the application */
            pPktNew = (*localDev.Config.pfcbRxPacket)
                                        (pdc->pd->hApplication,pPkt) ;
        }

        /* See if this was the last buffer */
        if( pdc->pDescRead == pDescAck )
            tmp = 0 ;

        /* Move the read pointer and decrement count */
        if( pdc->pDescRead == pdc->pDescLast )
            pdc->pDescRead = pdc->pDescFirst ;
        else
            pdc->pDescRead++ ;
        pdc->DescCount-- ;

        /* See if we got a replacement packet; if we do, we can immediately queue it */
        if( pPktNew )
        {
            /* Fill in the descriptor for this buffer */
            pDesc = pdc->pDescWrite ;

            /* Move the write pointer and bump count */
            if( pdc->pDescWrite == pdc->pDescLast )
                pdc->pDescWrite = pdc->pDescFirst ;
            else
                pdc->pDescWrite++ ;
            pdc->DescCount++ ;

            /* Supply buffer pointer with application supplied offset */
            pDesc->pBuffer   = pPktNew->pDataBuffer + pPktNew->DataOffset ;
#if !USE_EMAC_OPT
            pDesc->BufOffLen = localDev.PktMTU ;
            pDesc->PktFlgLen = EMAC_DSC_FLAG_OWNER ;
#endif
```

Example 15. Dequeue Receive Descriptor Function Code (continued)

```
            /* Push the packet buffer on the local descriptor queue */
            pqPush( &pdc->DescQueue, pPktNew ) ;
        }
    }

    /*
    // If we added descriptors, make the pNext of the last NULL, and
    // make the previous descriptor point to the new list we added.
    */
    if( pDescNewRxLast )
    {
        pDescNewRxLast->pNext = 0 ;

        /* Make the previous buffer point to us */
        if( pDescNewRxFirst == pdc->pDescFirst )
            pTemp = pdc->pDescLast ;
        else
            pTemp = pDescNewRxFirst-1 ;

        /*
        // If these pointers wrapped, the RX engine is stopped
        // Otherwise, tack the new list to the old
        */
        if( pTemp != pDescNewRxLast )
            pTemp->pNext = pDescNewRxFirst ;
    }
```

6.4 Transmit

The transmission of Ethernet packets is performed through the use of a buffer descriptor system where the application software or device driver describes the packet to send using one or more memory buffers descriptors. There is one descriptor for each noncontiguous block of memory in the packet (packet fragment). The buffer descriptor is a 16–byte memory structure that is stored in a 8K–byte memory space contained in the EMAC control module. The control module has space for up to 512 descriptors. You should be familiar with the EMAC operational overview and the detailed description of the transmit buffer descriptor fields, which can be found in the *TMS320C645x DSP Ethernet Media Access Controller (EMAC) / Management Data Input/Output (MDIO) User's Guide* (SPRU975).

As with the receive operation, there are a number of options for implementing the transmit operation on the EMAC hardware. The example code described here supports up to 8 different transmit channels. Each channel is allocated a static number of buffer descriptor slots from the EMAC control module memory block at initialization. The algorithm chosen for the example code is:

(512 less those required by receive) / number of transmit channels

Note that since a packet must fit entirely in the descriptor list in order to be sent, the maximum number of packet fragments that make up a packet can not exceed the total number of buffer descriptors allocated for a particular channel. For example, in a system that uses 64 buffer slots for receive and has eight transmit channels, each transmit channel would be allocated 56 buffer descriptor slots. Thus a single packet in such an environment could not contain more than 56 packet fragments. If only two transmit channels were used, each would have 224 buffer descriptors available. In environments where a static descriptor allocation does not yield acceptable results, a dynamic allocation method can be used.

In practice, there are usually more transmit descriptor slots available that are ever needed. However the software should be written to deal with transmit descriptor slot shortfalls. It is not necessary to have the transmit descriptor list as "deep" as receive because additional transmit packets can always be queued in software. Worst case for transmit is that there is a small delay in sending out the next packet, while the worst case for receive is a dropped packet.

Each transmit channel has its own channel descriptor structure. The structure is identical to that used for packet receive:

```
/*
// Transmit/Receive Descriptor Channel Structure
*/
typedef struct _EMAC_DescCh {
    struct _EMAC_Device *pd ;          /* Pointer to parent structure              */
    PKTQ                DescQueue ;    /* Packets queued as descriptors            */
    PKTQ                WaitQueue ; /* Packets waiting for transmit descriptors */
    uint                ChannelIndex ; /* Channel index 0-7                        */
    uint                DescMax ;   /* Maximum number of descriptors (buffers)  */
    uint                DescCount ;    /* Current number of descriptors            */
    EMAC_Desc           *pDescFirst ;  /* First descriptor location                */
    EMAC_Desc           *pDescLast ;   /* Last descriptor location                 */
    EMAC_Desc           *pDescRead  ;  /*  Location to read next descriptor    */
    EMAC_Desc *pDescWrite ; /* Location to write nest descriptor  */
} EMAC_DescCh ;
```

For a transmit channel, each descriptor refers to a full packet or a partial packet (packet fragment). For each buffer descriptor, there is a corresponding packet structure. The packet structures are kept in two queues. The `DescQueue` represents packets or packet fragments that are already represented by buffer descriptors in the channel. The `WaitQueue` is a queue of packet structures that are waiting to be placed into buffer descriptors.

The descriptors are tracked using the variables, `pDestFirst`, `pDestLast`, `pDescRead`, and `pDescWrite`. The `pDescFirst` and `pDescLast` pointers just point to the first and last descriptors in the fixed circular queue; these never change. The `pDescRead` pointer points to the next descriptor whose packet buffer is the next to be sent out on the network. The `pDescWrite` pointer points to the descriptor to use when adding the next packet to be transmitted.

It is helpful to consider how the packet buffers are represented in the code. The example code in Example 16 uses a structure of type `EMAC_Pkt` to define a packet. This structure has little to do with the EMAC hardware, but must be understood to follow the software examples. The structure and its related flags are defined below. Note that it is significantly similar to the descriptor format.

Example 16. Transmit Packets Example Code

```
typedef struct_EMAC_Pkt {
struct _EMAC_Pkt *pPrev ;               /* Previous record                          */
struct _EMAC_Pkt *pNext ;               /* Next record                              */
Uint8            *pDataBuffer ;         /* Pointer to data buffer                   */
Uint32           BufferLen ;            /* Physical length of buffer (read only)    */
Uint32           Flags ;                /* Packet flags                             */
Uint32           ValidLen ;             /* Length of valid data in buffer           */
Uint32           DataOffset ;           /* Byte offset to valid data                */
Uint32           PktChannel ;           /* Transmit channel/Priority 0-7 (SOP only) */
Uint32           PktLength ;            /* Length of packet (SOP only)              */
Uint32           PktFrags ;             /* Number of fragments in packet (SOP only) */

} EMAC_Pkt ;


/*

// Packet Buffer Flags set in Flags

*/
#define EMAC_PKT_FLAGS_SOP          0x80000000u       /* Start of packet    */
#define EMAC_PKT_FLAGS_EOP          0x40000000u       /* End of packet      */
```

6.4.1 Send Function

Since the packet send process starts with the `sendPacket()` function, we need to understand how the send function works in order to understand the rest. In some applications or drivers, it may not be necessary to support fragmented packets. For example, some TCP/IP stacks will never build a packet for transmission that spans more than one memory buffer. However, since fragmented packets are still somewhat common, the example software we show here does support them.

The code in Example 17 is taken from the packet send function in the example code. Much of the packet validation checking has been removed from this code. For purposes of sending the packet using the EMAC, the following operations are performed in the send function:

1. Make sure the first fragment of the packet has the SOP flag set in its flags member.
2. Count the number of packet fragments by parsing the packet until the EOP flag is found. This also verifies the correctness of the packet buffer chain. Note that only the first packet fragment can have the SOP flag set. This is also checked.
3. Get a pointer (in `pdc`) to the descriptor channel structure corresponding to the transmit channel specified by the caller.
4. Make sure the total number of fragments in the packet does not exceed DescMax; otherwise, the entire packet would never fit in the buffer descriptor list allocated for this channel.
5. Push the packet buffer(s) onto the `WaitQueue`. This is the queue for packer buffers waiting to be written out to the descriptor chain. At this point we do not know if the packet can be written or not. Even if it can, it must be placed in the queue behind any potential previously pending packets.
6. Call the `emacEnqueueTx()` function to remove as many packets as possible from the `WaitQueue` and write them into the buffer descriptor list.

The source code to implement this function is shown in Example 17. The `EMAC_Pkt` structure of the first fragment of the packet to send is pointed to by `pPkt`.

Example 17. Send Function Code

```
uint              fragcnt ;
EMAC_Pkt          *pPktLast ;
EMAC_DescCh       *pdc ;

/* Do some packet validation */
if( !(pPkt->Flags & EMAC_PKT_FLAGS_SOP) )
    return( EMAC_ERROR_BADPACKET );

/* Count the number of fragments in this packet */
fragcnt = 1 ;
pPktLast = pPkt ;
while( !(pPktLast->Flags & EMAC_PKT_FLAGS_EOP) )
{
    if( !pPktLast->pNext )
        return( EMAC_ERROR_INVALID ) ;
    pPktLast = pPktLast->pNext ;
    fragcnt++ ;

    /* At this point we cannot have another SOP */
    if( pPktLast->Flags & EMAC_PKT_FLAGS_SOP )
        return( EMAC_ERROR_INVALID ) ;

}

/* Get a local pointer to the descriptor channel */
pdc = &( pd->TxCh[pPkt->PktChannel] ) ;

/* Make sure this packet does not have too many fragments to fit */
if( fragcnt > pdc->DescMax )
    return( EMAC_ERROR_BADPACKET ) ;

/*
// Queue and packet and service transmitter
*/
pqPushChain( &pdc->WaitQueue, pPkt, pPktLast, fragcnt );
emacEnqueueTx( pdc) ;
```

6.4.2 Enqueue Transmit Descriptor Function

The `emacEnqueueTX()` function is pretty simple mostly because the work of structuring the packet buffers has already been done. The process for enqueuing a packet to the descriptor ring for transmit is:

1. Record the state of the descriptor set (first writable descriptor and the current count). The pointer to the first writable descriptor is saved so that it can be linked to the currently active list (if any) once descriptors for all waiting packets (or packet fragments) have been written. Unlike receive, we can not chain as we go because it is illegal to have a partial packet in the active transmit list at any given time. The save count tells us if the transmitter was running when we first began to add buffer descriptors.
2. Access the `WaitQueue` count to see if there are any packets waiting. We try to read all the packets from the `WaitQueue` and write their buffers into the descriptor list. If at any time there is no room in the descriptor list for all the fragments of the next waiting packet, we stop.
3. The number of packet fragments is known and part of the packet header. For each buffer in the packet, pop the packer header off `WaitQueue` and the fill in the descriptor list with the pointer to the packet (or packet fragment buffer). The next buffer descriptor to write is found in `pDescWrite`. The value of `pDescWrite` is then incremented.
4. When filling in the descriptor, the OWNER bit is added to all descriptors. Any SOP and EOP bits are also retained. On the SOP packet buffer, the total size of the packet is also written to the buffer descriptor.
5. The packet buffer head is then pushed onto the `DescQueue`. This queue is the holding spot for packet

buffers that currently occupy slots in the buffer descriptor list, and the two are always kept synchronized.

6. Once all the packets have been written to descriptors, or when there is no more room in the descriptor list, the process stops. Next, the list must be appended onto any previously existing list, or if there was no list, the new entries written become the active list.
7. Verify that new entries have been written. If so, check to see if there were previous entries. If there were previous entries, chain the descriptor before the first new descriptor written to the new list.
8. If there were new entries written, but there were no previous entries, then the new entries constitute a net transmit descriptor list for the channel in question. Start the transmitter by writing a pointer to the head of the new list (the saved `pDescOrg` value) to TX*n*HDP. The correct index to use is based on the transmit channel being processed.

The source code to implement this function is in Example 18.

Example 18. Enqueue Transmit Descriptor Function Code

```
static void emacEnqueueTx( EMAC_DescCh *pdc )
{
    EMAC_Desc        *pDescOrg,*pDescThis ;
    EMAC_Pkt         *pPkt ;
    uint             PktFrags ;
    uint             CountOrg ;
    volatile Uint32  *pRegAddr;

    /* Record the state of the descriptor set */
    pDescOrg = pdc->pDescWrite ;
    CountOrg = pdc->DescCount ;

    /* Try to post any waiting packets */
    while( pdc->WaitQueue.Count )
    {
        /* See if we have enough room for a new packet */
        pPkt = pdc->WaitQueue.pHead ;
        PktFrags = pPkt->PktFrags ;

        if( (PktFrags+pdc->DescCount) > pdc->DescMax )
            break ;

        /* The next packet will fit, post it */
        while( PktFrags )
        {
            /* Pop the next fragment off the wait queue */
            pPkt = pqPop( &pdc->WaitQueue ) ;

            /* Assign the pointer to "this" desc */
            pDescThis = pdc->pDescWrite ;

            /* Move the write pointer and bump count */
            if( pdc->pDescWrite == pdc->pDescLast )
                pdc->pDescWrite = pdc->pDescFirst ;
            else
                pdc->pDescWrite++ ;
                pdc->DescCount++ ;

            /*
            // If this is the last fragment, the forward pointer is NULL
            // Otherwise, this descriptor points to the next fragment's descriptor
            */

            if( PktFrags==1 )
                pDescThis->pNext = 0 ;
            else
                pDescThis->pNext = pdc->pDescWrite;
```

Example 18. Enqueue Transmit Descriptor Function Code (continued)

```
                pDescThis->pBuffer   = pPkt->pDataBuffer + pPkt->DataOffset ;
                pDescThis->BufOffLen = pPkt->ValidLen ;
            if( pPkt->Flags & EMAC_PKT_FLAGS_SOP )
                pDescThis->PktFlgLen = ((pPkt->Flags &
                                         (EMAC_PKT_FLAGS_SOP | EMAC_PKT_FLAGS_EOP)) |
                                         pPkt->PktLength|EMAC_DSC_FLAG_OWNER) ;
            else
                pDescThis->PktFlgLen = (pPkt->Flags & EMAC_PKT_FLAGS_EOP) |
                                         EMAC_DSC_FLAG_OWNER ;

            /* Enqueue this fragment onto the descriptor queue */
            pqPush( &pdc->DescQueue, pPkt ) ;
            PktFrags-- ;
        }
    }

    /* If we posted anything, chain on the list or start the transmitter */
    if( CountOrg != pdc->DescCount )
    {
        if( CountOrg )
        {
            /*
            // Transmitter is already running. Just tack this packet on
            // to the end of the list (we need to "back up" one descriptor)
            */
            if( pDescOrg == pdc->pDescFirst )
                pDescThis = pdc->pDescLast ;
            else
                pDescThis = pDescOrg - 1 ;
                pDescThis->pNext = pDescOrg ;
        }
        else
        {
            /* Transmitter is not running, start it up */
            pRegAddr = &EMAC_REGS->TX0HDP ;
        *(pRegAddr + pdc->ChannelIndex) = (Uint32)pDescOrg ;        }
    }
}
```

6.4.3 Deque Transmit Descriptor Function

Once the EMAC has finished transmitting a packet, it returns the packet buffers associated with the packet to the software application in much the same way the newly received packets are indicated. The `emacDequeueTX()` function removes the completed transmit buffers, returning the buffers to the software application, and marking the descriptors from transmit channel to the descriptor free list for use for more transmit operations.

To understand this function better, it is important to know what happens during a device interrupt. The ISR code relating to the receive operation is:

```
pRegAddr = &EMAC_REGS->TX0CP;

/* Look for transmit interrupt (channel 0-max) */
for( tmp=0; tmp<pd->Config.TxChannels; tmp++ )
    if( intflags & CSL_FMK( EMAC_MACINVECTOR_TXPEND, 1<<tmp ) )
    {
        Desc = *(pRegAddr + tmp) ;
        *(pRegAddr + tmp) = Desc ;

        emacDequeueTx( &pd->TxCh[tmp], (EMAC_Desc *)Desc ) ;
     }
```

For each active channel in the system, the TXPEND field of the MACINVECTOR register is examined to see if the particular channel has seen new activity. Next, the last descriptor to process in the given channel can be read from TX*n*CP register, where the index is based on the channel number. This is also the register where we write the value of the last descriptor processed. Since the `emacDequeueTx()` function processes all descriptors up to the one that it is passed, the transmit interrupt can be immediately acknowledged by writing the value back to the TX*n*CP register. Finally, the `emacDequeueTx()` function is called with a pointer to the transmit descriptor channel structure and a pointer to the last descriptor to service.

The `emacDequeueTx()` function is shown in Example 19. The functions it needs to perform are:

1. The next descriptor to process is always available at the `pDescRead` pointer. The flags for that descriptor are read. The only flag that is important here is the EMAC_DSC_FLAG_EOQ flag that is checked at the end of the loop.
2. The `EMAC_Pkt` structure corresponding to the descriptor is recovered from the `DescQueue`. This buffer is returned to the application by use of a callback function.
3. If this is the last descriptor to process `(pDescRead == pDescAck)`, then a flag is set to prevent the loop from executing again.
4. The `pDescRead` pointer is incremented and the `DescCount` is decremented.
5. Continue until all the descriptors have been processed up to and including that indicated by the caller (in this case the ISR).
6. If the last descriptor processed had the EMAC_DSC_FLAG_EOQ flag set in its flags field, this means that the EMAC interpreted the descriptor as being the last in the descriptor chain (its next pointer was NULL). This occurs if there are no more packets to transmit, or if any newly chained packets were chained on after the transmitter stopped. If the EOQ flag was set and there are more packet descriptors waiting, then restart the transmitter by posting the head of the descriptor list `(pDescRead)` to TX*n*HDP.
7. As a final step, since descriptor entries have been freed, if there are more transmit packets waiting on the `WaitQueue` (waiting to be added to the descriptor list), then call the `emacEnqueueTX()` function to enqueue these packets.

The source code to implement this function is in Example 19.

Example 19. Dequeue Transmit Descriptor Function Code

```
static void emacDequeueTx( EMAC_DescCh *pdc, EMAC_Desc *pDescAck )
{
    EMAC_Pkt     *pPkt ;
    uint         i,j = (uint)pdc->pDescRead ;
    Uint32       PktFlgLen ;
    Volatile     uint32 * pRegAddr ;


    /* Get the status of the ACK descriptor */
    PktFlgLen = pDescAck->PktFlgLen ;

    /* Calculate the new "Read" descriptor */
    if( pDescAck == pdc->pDescLast )
        pdc->pDescRead = pdc->pDescFirst ;
    else
        pdc->pDescRead = pDescAck+1 ;

    i = (uint)pdc->pDescRead ;

    /* Turn i into a descriptor count */
    if( j < i )
        i = (i-j)/sizeof(EMAC_Desc);
    else
        i = pdc->DescMax - ((j-i)/sizeof(EMAC_Desc));

    pdc->DescCount-= i ;
```

Example 19. Dequeue Transmit Descriptor Function Code (continued)

```
    /* Pop & free buffers until the last descriptor */
    while(i--)
    {
        /* Recover the buffer and free it */
        pPkt = pqPop( &pdc->DescQueue ) ;
        if( pPkt )
            (*localDev.Config.pfcbFreePacket)(pdc->pd->hApplication,pPkt) ;
    }

    /* If the transmitter stopped and we have more descriptors, then restart */
    if( (PktFlgLen & EMAC_DSC_FLAG_EOQ) && pdc->DescCount ) {
        pRegAddr = &EMAC_REGS->TX0HDP ;
        *(pRegAddr + pdc->ChannelIndex) = (Uint32)pdc->pDescRead;
    }

    /* Try to post any waiting TX packets */
    if( pdc->WaitQueue.Count )
        emacEnqueueTx( pdc ) ;
}
```

6.5 EMAC Interface Configuration

The EMAC on C645x devices has four interfaces to transfer data to a PHY. MII and RMII interface support 10/100 Mbps speeds only, while GMII and RGMII support 10/100/1000 Mbps. Only one interface is in use at any given time; this setting is programmed in hardware, and cannot be altered in software.

This example driver reads the MACSEL field of the DEVSTAT register, which is located at device level, to determine the interface used, and to configure it appropriately. The EMAC related configuration for each interface is done in the `EMAC_timerTick()` function, which gets called every time the PHY negotiates a new link.

The main settings that the EMAC negotiates with the PHY are speed and duplex mode. Normally, EMAC uses the FULLDUPLEX bit in MACCONTROL register for duplex mode configuration, which signifies that full-duplex is to be used when set, and half-duplex when not set. The speed selection is transmitted from the application to the PHY, which is then configured appropriately.

There are some interface specific configurations needed as well, which are detailed below. The code for the configuration of each interface, as directed by the duplex and speed required by the application, may appear as in Example 20.

6.5.1 RMII Interface Configuration

Duplex and speed mode for the RMII interface are set from the RMIIDUPLEXMODE and RMIISPEED fields of the EMAC's MACCONTROL register. The state of the FULLDUPLEX field does not have any effect when using the RMII interface.

Once the speed and duplex mode are set in the MACCONTROL register, RMII interface needs to be taken out of reset, by clearing bit 18 of the device level EMACCFG register.

6.5.2 Gigabit Interfaces Configuration

When the application sets the MDIO flags to 1000 Mbps, and full duplex (MDIO_MODEFLG_FD1000), the GIG bit in the MACCONTROL register needs to be set. Since the gigabit speed is supported only in full-duplex mode, the FULLDUPLEX field of MACCONTROL also needs to be set.

For the RGMII interface to work with any physical device, including those that do not support in-band signaling, the RGMII field in the MACCONTROL register is reset. This causes the RGMII logic to operate in forced link mode.

Example 20. Interface Configuration Code

```
if( linkStatus == MDIO_LINKSTATUS_FD10 ||
    linkStatus == MDIO_LINKSTATUS_FD100 ||
    linkStatus == MDIO_LINKSTATUS_FD1000 )
{
    CSL_FINST( EMAC_REGS->MACCONTROL, EMAC_MACCONTROL_FULLDUPLEX, ENABLE );

    if ( macsel == CSL_DEV_DEVSTAT_MACSEL_RMII )
       CSL_FINST( EMAC_REGS->MACCONTROL, EMAC_MACCONTROL_RMIIDUPLEXMODE, FULLDUPLEX ) ;
}
else
{
    CSL_FINST( EMAC_REGS->MACCONTROL, EMAC_MACCONTROL_FULLDUPLEX, DISABLE );
    if ( macsel == CSL_DEV_DEVSTAT_MACSEL_RMII )
        CSL_FINST( EMAC_REGS->MACCONTROL, EMAC_MACCONTROL_RMIIDUPLEXMODE, HALFDUPLEX );
}

if( linkStatus == MDIO_LINKSTATUS_FD1000 )
    CSL_FINST( EMAC_REGS->MACCONTROL, EMAC_MACCONTROL_GIG, ENABLE ) ;

if(( (linkStatus == MDIO_LINKSTATUS_HD10 ) ||
    ( linkStatus == MDIO_LINKSTATUS_FD10 )) &&
    ( macsel == CSL_DEV_DEVSTAT_MACSEL_RMII ))
        CSL_FINST( EMAC_REGS->MACCONTROL, EMAC_MACCONTROL_RMIISPEED, 2_5MHZ ) ;

if(( (linkStatus == MDIO_LINKSTATUS_HD100 ) ||
    ( linkStatus == MDIO_LINKSTATUS_FD100 )) &&
    ( macsel == CSL_DEV_DEVSTAT_MACSEL_RMII ))
        CSL_FINST( EMAC_REGS->MACCONTROL, EMAC_MACCONTROL_RMIISPEED, 25MHZ ) ;

/* Take RMII out of reset */
if( macsel == CSL_DEV_DEVSTAT_MACSEL_RMII )
    CSL_FINST(DEV_REGS->EMACCFG, DEV_EMACCFG_RMIIRST, RELEASE );

/* Put RGMII in forced link mode */
if( macsel == CSL_DEV_DEVSTAT_MACSEL_RGMII )
    CSL_FINST(EMAC_REGS->MACCONTROL, EMAC_MACCONTROL_RGMIIEN, DISABLE ) ;
```

6.6 Interrupt Processing

The interrupt signals on the EMAC and MDIO are combined into a single interrupt inside the EMAC control module. The interrupt is used to signal the application or device driver that work needs to be done on the EMAC or MDIO.

All the interrupt signals are combined in the EMAC control module, and this combined set is also fed back into the EMAC module and can be examined by software by reading the MACINVECTOR register. Note that this register represents the masked set of interrupt bits. If an interrupt is not enabled in its corresponding register on the EMAC or the MDIO, then its interrupt bit in the MACINVECTOR register will never be set.

The example software does not use interrupts on the MDIO module. This is because the same operations can be performed as a timer event driven state machine. There is no need for real time caliber response times in servicing MDIO.

6.6.1 Interrupt Deferral

Depending on the run–time environment, an application or device driver may or may not do any actual processing in its ISR. For example, consider a system that calls a function like `netISR()`, where the job of the function is just to turn off the device ISR and return TRUE if the device generated the interrupt, and FALSE if it did not. In a system like this, another work function would be called to actually do the ISR servicing, but not at interrupt time. An implementation of `netISR()` may look like:

```
int netISR()
{
    Uint32 intflags ;

    /* Read the interrupt cause */
    if( (intflags = EMAC_REGS->MACINVECTOR ) != 0 )
    {
        /* Disable EMAC/MDIO interrupts in the control module */
        CSL_FINST( ECTL_REGS->EWCTL, ECTL_EWCTL_INTEN, DISABLE ) ;

        /* Tell the caller it was our interrupt */
        return( 1 );
    }

    /* Tell the caller it was not our interrupt */
    return( 0 ) ;
}
```

This function disables the device interrupt if it is going to return TRUE. The interrupt is then re-enabled once processing is done.

When interrupt pacing is used (programmed using the EWINTTCNT register), the interrupt pace counter does not start counting down until interrupts are re-enabled in EWCTL. Thus, if a static pace time is used (where the value of EWINTTCNT is not changed), the delay from the time `netISR()` is called to the time the interrupts are re-enabled in EWCTL can alter interrupt timing. If a static count in EWINTTCNT is used, and the interrupts are certain to be serviced in that amount of time allotted via this register, then it is acceptable to rewrite the previous function as follows:

```
int netISR()
{
    Uint32 intflags ;

    /* Read the interrupt cause */
    if( (intflags = EMAC_REGS->MACINVECTOR ) != 0 )
    {

        /* Disable EMAC/MDIO interrupts in the control module */
        CSL_FINST( ECTL_REGS->EWCTL, ECTL_EWCTL_INTEN, DISABLE ) ;

        /* Start counter to re-Enable EMAC/MDIO interrupts */
        CSL_FINST( ECTL_REGS->EWCTL, ECTL_EWCTL_INTEN, ENABLE ) ;

        /* Tell the caller it was our interrupt */
        return( 1 ) ;
    }

    /* Tell the caller it was not our interrupt */
    return( 0 ) ;
}
```

Keep in mind that this is only one approach to handling interrupts. In the example code, the interrupt processing is done directly by the ISR, and not deferred.

6.6.2 Interrupt Handling

As can be seen in the definition of the MACINVECTOR register, there are six reasons the EMAC control module interrupt can fire. They are listed in Table 1.

Table 1. Reasons EMAC Control Module Generates Interrupt

Name	Description
USERINT	The MDIO has completed a read or write access to a PHY control register.
LINKINT	The link status of a PHY monitored by the MDIO has changed.
HOSTPEND	A host interrupt is pending on the EMAC. This signifies an error condition.
STATPEND	One of the EMAC statistics registers is in danger of overflow (has its MSB set).
RXPEND	One or more of the 8 receive channels needs servicing.
TXPEND	One or more of the 8 transmit channels needs servicing.

The sample code does not use either the USERINT or LINKINT interrupt signals. The USERINT signal is only good for accessing PHY configuration registers as a background task through the MDIO module. Although accessing PHY configuration register does take many cycles, it is only done at initialization, and does not need to be a general background task. The LINKINT interrupt generates when the link status changes on a monitored PHY. However, since link status can take up to 3 seconds to change, it is perfectly acceptable to poll for this condition. An interrupt is not necessary. This is discussed more in Section 5.

An excerpt from the sample code interrupt processing is shown in Example 21. This processing is independent of the DSP interrupt. The DSP interrupt is handled in the normal fashion. This interrupt processing code performs:

- Disable device interrupts by writing the EWCTL register. Note that this serves two purposes. It drives the interrupt signal low, so that the next rise triggers an interrupt on the DSP (that is edge triggered). Also, disabling then re-enabling interrupts in the EWCTL register restarts the pace counter (when used) that determines when another interrupt can be generated to the DSP.
- The MACINVECTOR register is read into a temporary register. This value contains flags representing the state of every possible interrupt source on the EMAC and MDIO modules.
- When the HOSTPEND bit is set, the EMAC has encountered an error caused by the host software. The error status is reported to the application using a callback so that the application can correct the problem and reset the device.
- When the STATPEND bit is set, one of the EMAC statistics registers is in danger of overflow. Thus, the software calls a function to read and reset all the statistics values and keep a soft copy locally. It then notifies the application using a callback so that the application can read the new statistics values. However, since the EMAC statistics registers have already been read and cleared, the sample code does not need to rely on the application responding to the callback to clear the interrupt condition.
- Next, check for each of the eight possible TXPEND bits, depending on how many transmit channels are in use. For each transmit channel requiring servicing, service it in accordance with the procedure outlined in Section 6.4.
- Next, check for each of the eight possible RXPEND bits, depending on how many receive channels are in use. This sample code only uses a single receive channel, so there is no for–next loop. If the receive channel requires servicing, service it in accordance with the procedure outlined in Section 6.3.
- As a final step, interrupts are re-enabled by writing the EWCTL register. If an interrupt is still pending, this causes another rising edge and retriggers the DSP interrupt. Interrupts are rearmed immediately, if interrupt pacing is not used. If a count is programmed into the EWINTTCNT register, then interrupts are not rearmed until that value of peripheral clock cycles have expired. The peripheral clock is CPUclock/6.

The source code to perform this operation is in Example 21. The function `emacUpdateStats()` is used to read the statistics and then clear the statistics register. This is done by writing back the value for each statistic read to its corresponding register. The registers are write–to–decrement, so no stats are lost.

Example 21. Interrupt Processing Code

```
Uint32          intflags,Desc ;
uint            tmp ;
volatile Uint32 *pRegAddr ;

/* Disable EMAC/MDIO interrupts in the control module */
CSL_FINST( ECTL_REGS->EWCTL, ECTL_EWCTL_INTEN, DISABLE ) ;

/* Read the interrupt cause */
intflags = EMAC_REGS->MACINVECTOR ;

/* Look for fatal errors first */
if( intflags & CSL_FMK( EMAC_MACINVECTOR_HOSTPEND, 1 ) )
{
    /* Read the error status - we'll decode it by hand */
    pd->FatalError = EMAC_REGS->MACSTATUS ;

    /* Tell the application */
    (*localDev.Config.pfcbStatus)(pd->hApplication) ;

    /* Return with interrupts still disabled in the control module */
    Return ;
}

/* Look for statistics interrupt */
if( intflags & CSL_FMK( EMAC_MACINVECTOR_STATPEND, 1 ) )
{
    /* Read the stats and write-decrement what we read */
    /* This is necessary to clear the interrupt */
    emacUpdateStats( pd ) ;

    /* Tell the application */
    (*localDev.Config.pfcbStatistics)(pd->hApplication) ;    /*
}

/* Look for transmit interrupt (channel 0-max) */
pRegAddr = &EMAC_REGS->TX0CP ;
for( tmp=0; tmp<pd->Config.TxChannels; tmp++ )
    if( intflags & CSL_FMK( EMAC_MACINVECTOR_TXPEND, 1<<tmp ) )
    {
        Desc = *(pRegAddr + tmp) ;
        *(pRegAddr + tmp) = Desc ;
        emacDequeueTx( &pd->TxCh[tmp], (EMAC_Desc *)Desc ) ;
    }

/* Look for receive interrupt (channel 0) */
if( intflags & CSL_FMK( EMAC_MACINVECTOR_RXPEND, 1<<0 ) )
{
    Desc = EMAC_REGS->RX0CP ;
    EMAC_REGS->RX0CP = Desc ;
    emacDequeueRx( &pd->RxCh, (EMAC_Desc *)Desc ) ;
}

/* Enable EMAC/MDIO interrupts in the control module */
CSL_FINST( ECTL_REGS->EWCTL, ECTL_EWCTL_INTEN, ENABLE ) ;
```

6.7 Shutdown and Restarts

A shutdown is necessary to make sure the EMAC does not continue to access DSP memory (or generate interrupts) after the device is closed. Also, a graceful shutdown is the first stage of a proper device restart.

The example software discussed in this document implements device restart as a call to its `close()` function followed by a second call to `open()`. The open operation and device initialization steps are discussed earlier in this chapter. This section describes the device close procedure. The steps for shutting down the device are:

1. Disable device interrupts by writing the EWCTL register. This prevents further interrupts from the device. It is assumed that the DSP interrupt to which the EMAC control module is mapped has also been masked, and any pending condition cleared after this close function is complete (and most likely remain masked).
2. Initiate a teardown of each channel in use by using the RXTEARDOWN and TXTEARDOWN registers. In the example code, there is only one receive channel, but up to eight transmit channels.
3. When the HOSTPEND bit is set in the ISR, a fatal error occurs. If this close operation was started after a fatal error, then the teardown operations will never complete. Thus, the fatal error status of the device is checked before waiting for teardown to complete.
4. If no fatal error occurred, then the software should wait for the shutdown operation to complete on each channel by reading the RX*n*CP and TX*n*CP registers for each corresponding channel. The register reads FFFF FFFCh when the teardown operation is complete. This value is then written back to RX*n*CP or TX*n*CP by the software to acknowledge the teardown completion indication.
5. Clear the MACCCONTROL, RXCONTROL, and TXCONTROL registers.
6. Finally, clean up the software environment. In the example code, this involves releasing all memory buffers back to the application using a callback function.

The source code to implement this operation is shown in Example 22.

Example 22. Device Shutdown Example Code

```
/* Disable EMAC/MDIO interrupts in the control module */
CSL_FINST(ECTL_REGS->EWCTL, ECTL_EWCTL_INTEN, DISABLE ) ;

/* Teardown receive */
EMAC_REGS->RXTEARDOWN = 0 ;

/* Teardown transmit channels in use */
for( i=0; i<pd->Config.TxChannels; i++)
    EMAC_REGS->TXTEARDOWN = i ;

/* Only check teardown status if there was no fatal error */
if( !pd->FatalError )
{
    /* Wait for the teardown to complete */
    for( tmp=0; tmp!=0xFFFFFFFC; tmp=EMAC_REGS->RX0CP ) ;

    EMAC_REGS->RX0CP = tmp ;
    pRegAddr = &EMAC_REGS->TX0CP ;

    for( i=0; i<pd->Config.TxChannels; i++ )
    {
        for( tmp=0; tmp!=0xFFFFFFFC; tmp=*(pRegAddr + i) ) ;
        *(pRegAddr + i) = tmp ;
    }
}

/* Disable RX, transmit, and clear MACCONTROL */
 CSL_FINST(EMAC_REGS->TXCONTROL, EMAC_TXCONTROL_TXEN, DISABLE ) ;
 CSL_FINST(EMAC_REGS->RXCONTROL, EMAC_RXCONTROL_RXEN, DISABLE ) ;
 EMAC_REGS->MACCONTROL = 0;

/* Free all receive buffers */
while( pPkt = pqPop( &pd->RxCh.DescQueue ) )
```

Example 22. Device Shutdown Example Code (continued)

```
    (*pd->Config.pfcbFreePacket)(localDev.hApplication, pPkt) ;

/* Free all transmit buffers */
for( i=0; i<pd->Config.TxChannels; i++)
{
    while( pPkt = pqPop( &pd->TxCh[i].DescQueue ) )
        (*pd->Config.pfcbFreePacket)(localDev.hApplication, pPkt);
    while( pPkt = pqPop( &pd->TxCh[i].WaitQueue ) )
        (*pd->Config.pfcbFreePacket)(localDev.hApplication, pPkt);
}
```

7 Example Applications

The sections covered so far described the EMAC low level driver, and highlighted the implementation of some of its core APIs. This section discusses how these APIs can be used in an application. Several example applications are provided, highlighting different use cases for the driver. The applications showing how to use the low-level EMAC driver described in this document are DSP/BIOS based, and require the following development tools:

- Code Composer Studio™ 3.2
- DSP/BIOS™ 5.21

The register layer for EMAC and DEV CSL has been used in the development of the driver and example applications. The relevant CSL files are included with the code, under the `Include\` directory.

7.1 Enabling the EMAC/MDIO Peripheral

When the device is powered on, the EMAC peripheral is disabled, and no reads/writes can be made to its registers. Prior to EMAC-specific initialization, the EMAC must be enabled.

EMAC/MDIO is enabled through the chip level module state control register 0 (MDCTL0) and module status register 0 (MDSTAT0). For detailed information on the programming sequence, see the device-specific data manual. This sequence will enable the EMAC/MDIO peripheral, and the register values are reset to default.

The example applications use a common DSP/BIOS user initialization function, `gblUserInit()` to enable the peripheral. This function is an application level function, called by all the example applications, and is located in `\src\emac_init.c` file. After this function is executed, the EMAC may be initialized, and data transfer may be initiated.

7.2 Loopback Test

The example under `C6455_emac_loopback\` directory is the most basic, and it can serve as a quick hardware check. It transfers packets of various sizes in loopback mode, and checks for correctness of data received. This example can be used with several levels of loopback:

- EMAC internal loopback, when LOCAL_LOOPBACK constant is set to 1.
- Loopback internal to the PHY, when PHY_LOOPBACK constant is set to 1. LOCAL_LOOPBACK needs to be set to 0; otherwise the loopback will be done at the internal EMAC level.
- Loopback external to the PHY. When this mode is used, both LOCAL_LOOPBACK and PHY_LOOPBACK are set to 0, and a loopback connector needs to be placed in RJ-45. Note that a loopback connector with 8 wires needs to be used when running the example at 1000 Mbps speed (flag MDIO_MODEFLG_FD1000 is set in the main application function).

The loopback example main file `loopback_main.c` is used to show how to transfer unicast packets. This is the default file included in the project. In the same directory, file `loopback_main-multicast.c` shows how to install a multicast list, and transfer multicast packets.

7.3 Communicating with a PC: Echo Example

The example application under C6455_emac_echo\ directory was included to show how to transmit packets between a PC and a C645x device, using the sample driver APIs described in this document. The DSP side plays the role of an echo server: it sends back anything that it receives from the PC. On the PC side, the Windows application UDPFLOOD.EXE is used to send a stream of packets to the DSP.

Connect the DSP to the PC either with a cross-over cable, or connect both to a router or switch. Follow the steps below to run this example:

1 Load the c6455_emac_echo.OUT file from the Debug\ directory and run it. The program will print a few status lines, and then the number of packets it has echoed.

2 At a DOS prompt, choose an IP address that you can use for testing purposes; it must be an address on the local subnet. Add that address to the local ARP table of the PC. The MAC address of the DSP is set to 00-01-02-03-04-05 for this program. So, if you are choosing an IP address of 192.168.1.3, you need to enter this command at the DOS prompt:

```
C:\>arp -s 192.168.1.3 00-01-02-03-04-05
```

3 Run the UDPFLOOD.EXE program included in the c6455_emac_echo\ directory with the IP address you have chosen. For example:

```
C:\>udpflood.exe 192.168.1.3
```

4 The number of packets sent to the DSP is printed on the screen. To stop the UDP program, press <ENTER>.

7.4 Connecting Two DSPs: Send/Receive Example

This example is similar to the Echo example in Section 7.3, except that the data is transferred between two DSPs. You can run this application between a DSK baseboard and a mezzanine card, between two DSK baseboards, or between two mezzanine cards.

The server side of the example is located under C6455_emac_recv\ directory, and waits in a loop for the client to send packets. When it receives data packets, it changes their destination MAC address to match that of the sender, and it echoes them back. The client side, located under c6455_emac_send\ directory, connects to the receiver, and then sends packets of various sizes to the server, in an infinite loop.

To run this example, follow these steps:

1. Connect the two DSPs together using either a cross-over cable, or connect each of them to a switch via straight cables. Make sure you use the same speed and duplex settings for the two EMACs, when not using auto-negotiation. For example, you cannot set one EMAC to run at 1000 Mbps in full duplex mode, and the other at 100 Mbps in half duplex mode.
2. Load and run first the receive program.
3. Immediately after starting the receiver, load and run the sender program.
4. The receiver will print out the number of packets echoed. The sender will also print the packets sent and then received back.

7.5 Benchmarking Example

The example under C6455_emac_benchmark\ directory shows a method to calculate the CPU load when EMAC transmits at full wire rate, for packets of various sizes.

When benchmarking a gigabit interface, use the benchmark_main_bench_GIG.c file; this is the default included in the project. For a 100 Mbps interface, use benchmark_main_bench_100.c. When using the file for the gigabit interfaces, make sure that you have the USE_JUMBO_PKT flag set to 1 in C6455_emac.c file. Otherwise, you will get an error when the application tries to send a packet bigger than the Ethernet standard packet (1514).

The benchmark example transfers data via PHY internal loopback, between different locations in memory. The default source and destination for packets is L2 memory, with the code executing from internal memory as well. The buffers descriptors are placed in the EMAC Control module RAM memory. To put the descriptors in the internal memory, partition the internal memory in such a way that the second half is reserved for the buffer descriptors, and then change this code in the `EMAC_open()` function:

```
    /* Pointer to first descriptor to use on RX */
    pDesc = (EMAC_Desc *)_EMAC_DSC_BASE_ADDR;

to
    pDesc = (EMAC_Desc *)_EMAC_DSC_BASE_ADDR_L2;
```

The descriptors location is restricted to internal memory and EMAC control module internal RAM. When they are placed in external memory the throughput results are not satisfactory.

8 Throughput Benchmarks

The throughput numbers below were obtained by running the benchmark application described in Section 7.5, on a 1 GHz C645x device. Some of the configuration options chosen for these benchmarks include:

- The EMAC receive packet optimization is turned on. See Section 6.1 for more details on what this optimization entails.
- The data was transferred via PHY internal loopback. Measurements did not show any difference from when an external loopback plug was used.
- The code is located in internal memory for all the benchmarks. The location of the data varies.

8.1 MII and RMII Interfaces

The CPU load corresponding to a full wire rate of 100 Mbps is shown in Table 2.

Table 2. 100Mbps Throughput With Data in Internal Memory

		CPU Load (%)	
Packet Size (Bytes)	Packets/Second	Descriptors in EMAC Internal RAM	Descriptors in L2
64	144809	36.1	34.8
128	84459	21.8	21.0
256	45289	13.8	13.4
512	23496	8.2	7.9
1024	11973	6.0	5.8
1518	8127	4.0	3.9

Table 3 gives the CPU load when data is transmitted at 100 Mbps wire rate, with buffer descriptors placed in EMAC Control Module internal RAM, and data buffers in external memory (DDR2).

Table 3. 100Mbps Throughput With Data in DDR2

		CPU Load (%)	
Packet Size (Bytes)	Packets/Second	256K Cache	64K Cache
64	144809	74.0	73.1
128	84459	44.2	41.7
256	45289	26.5	21.2
512	23496	15.6	10.1
1024	11973	10.1	4.1
1518	8127	6.7	0.3

8.2 GMII and RGMII Interfaces

The CPU load corresponding to a full wire rate of 1000 Mbps is shown in Table 4.

Table 4. 1000Mbps Throughput With Data in Internal Memory

		CPU Load (%)	
Packet Size (Bytes)	Packets/Second	Descriptors in EMAC Internal RAM	Descriptors in L2
256	452898	46.5	48.9
512	234962	45.6	53.8
1024	119731	29.7	28.8
1518	81274	20.8	20.3
4096	30339	9.7	10.3
10240	12183	5.4	5.3

Table 5 gives the CPU load when data is transmitted at full wire rate, with buffer descriptors placed in EMAC Control Module internal RAM, and data buffers in external memory (DDR2).

Table 5. 1000Mbps Throughput With Data in DDR2

		CPU Load (%)	
Packet Size (Bytes)	Packets/Second	256K Cache	64K Cache
256	452898	81.8	82.5
512	234962	80.5	80.1
1024	119731	61.2	60.0
1518	81274	41.8	39.1
4096	30339	19.6	13.6
10240	12183	9.7	4.5

9 References

1. *TMS320C645x DSP Ethernet Media Access Controller (EMAC) / Management Data Input/Output (MDIO) User's Guide* (SPRU975)
2. *TMS320C6455 Fixed-Point Digital Signal Processor Data Manual* (SPRS276)

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale (www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265