Freescale Semiconductor
Application Note

Document Number: AN3099
Rev. 0, 11/2006

MSC711x Ethernet Quick Start

by *Dejan Minic*
NCSG
Freescale Semiconductor, Inc.
Austin, TX

This application note covers code examples for programming the MSC711x Fast Ethernet Controller (FEC) in the MII mode. The MSC711x devices that support Ethernet are the MSC7113, the MSC7116, and the MSC7119. A brief introduction of the MSC711x Ethernet Controller is given first. Then code examples are presented for each of three different MII modes.

Contents

1 MSC711x Ethernet Controller Introduction

The FEC supports 10/100 Mbps Ethernet as defined by **IEEE** Std. 802.3™. It has two MAC-PHY interfaces: the media independent interface (MII) and the reduced MII (RMII), which provides MII functionality on a reduced pin count (10 instead of 18), and a 7-wire interface mode. A media access controller (MAC) handles the MII interface FIFOs and DMA functionality and an MII gasket (MIIGSK) module supports the RMII interface. An FEC RISC microcontroller manages DMA buffer descriptors (BDs), minimizing processor usage. Also, a management information base (MIB) module tracks network activity on the MAC-PHY interface.

1.1 Features

FEC features are as follows:

- Designed to comply with **IEEE** Stds. 802.3™, 802.3u™, 802.3x™, and 802.3ac™.
- Internal receive and transmit FIFOs and a FIFO controller.
- Direct access to internal MSC7119 memories via its own DMA controller.
- Support for the 10/100 Mbps media independent interface (MII)
- Support for the 10/100 Mbps reduced media independent interface (RMII).
- Support for the 10 Mbps 7-wire interface.
- Full duplex (200 Mbps throughput with a minimum system clock rate of 25 MHz) and half duplex operation (100 Mbps throughput).
- Programmable maximum frame length supports **IEEE** Std. 802.1™ VLAN tags and priority.
- Retransmission from transmit FIFO following a collision.
- CRC generation and verification for inbound and outbound packets.
- Automatic internal flushing of the receive FIFO for runt receive frames and receive frames rejected by address recognition.
- **IEEE** Std. 802.3 full duplex flow control.
- Address recognition including promiscuous, broadcast, individual address. hash/exact match, and multicast hash match:
  - — Frames with broadcast address can be always accepted or always rejected
  - — Exact match for single 48-bit individual (unicast) address
  - — 64-bit hash check of individual (unicast) addresses.
  - — 64-bit hash check of group (multicast) addresses.

1.2 FEC Architecture

Figure 1 shows the FEC block diagram with the network depicted at the bottom of the diagram. The IPBus and interrupt interfaces comply with V2 of the IPBus specification, and the AHB master interface connected to the AMENT bus complies with Rev. 2.0 of the AHB-Lite specification. The external Ethernet interfaces are designed to comply with industry and **IEEE** 802.3 standards.

The RISC-based descriptor controller, shown in Figure 1, performs the following functions:

- Initializes the internal registers not initialized by the user or hardware.
- Controls the DMA channels at a high level, initiating DMA data transfers.
- Interprets buffer descriptors (BDs).
- Provides address recognition for receive frames.
- Generates random numbers for the transmit collision back-off timer.

The RAM is the focal point of all FEC data flow. The RAM is divided into transmit and receive FIFOs with a boundary that is programmed in the FIFO receive start register (FRST). User data flows to/from the DMA unit from/to the receive/transmit FIFOs. Transmit data flows from the transmit FIFO into the transmit block, and receive data flows from the receive block into the receive FIFO.

The bus controller selects the TBus master on each clock. All modules receive their control information and provide status information over the TBus.

The user controls the FEC by writing through the slave interface (SIF) module into control registers in each block. The control and status registers provide global control, such as Ethernet reset and enable and interrupt handling.

The MII block provides a serial channel for control/status communication with the external physical layer device (transceiver). This serial channel consists of the MDC (clock) and MDIO bidirectional data lines of the MII interface.

The multiple-channel DMA unit allows transmit data, transmit descriptor, receive data, and receive descriptor accesses to run independently.

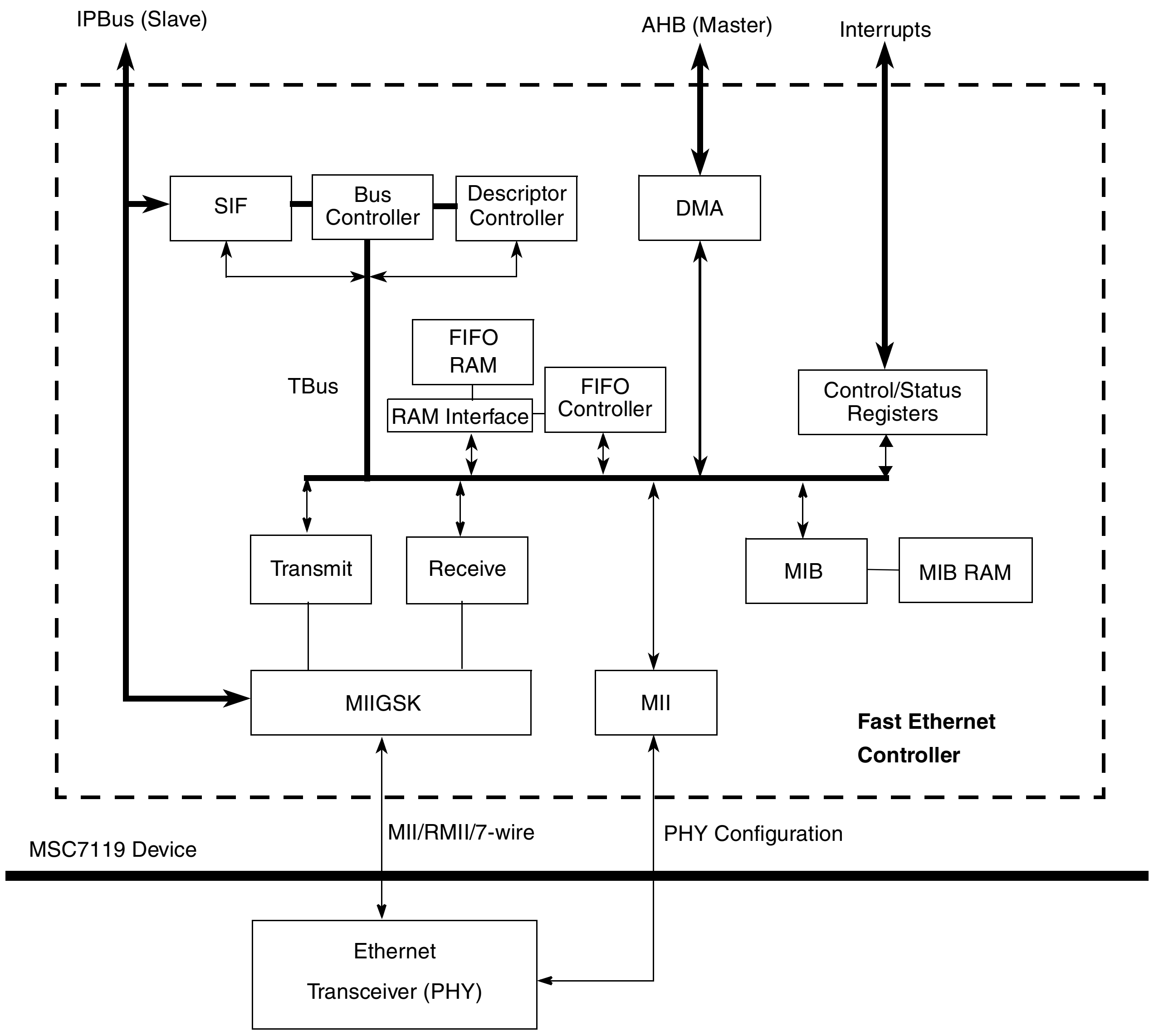


Figure 1. MSC7119 Fast Ethernet Controller Architecture

The transmit and receive blocks provide the Ethernet MAC functionality. Internal to these blocks are clock domain boundaries between the system clock and the network clocks.

The management information base (MIB) maintains counters for a variety of network events and statistics. It is not necessary for FEC operation but provides valuable counters for network management. The counters are remote monitoring (RMON) RFC 1757 Ethernet Statistics group and some **IEEE** Std. 802.3 counters.

The MIIGSK converts the data path portion of the MII interface to the RMII interface, reducing the MII data path pin count from 16 to 8 pins.

2 Description of the Ethernet Modes

This application note covers three MII Ethernet examples. Start with the first example, because each example builds on knowledge from the previous example. The three Ethernet mode examples are as follows:

1. MII internal loop with polling
2. MII internal loop with interrupts
3. MII external loop with interrupts

3 MII Internal Loop with Polling

All three examples use the predefined data packet, which is used for Tx and Rx Ethernet transmission. Consider code in the data.c file.

```
#pragma data_seg_name ".enetdata"
/*---------------------------------------------------*/
/* Receive and Transmit Buffer Descriptors */
/*---------------------------------------------------*/
NBUF EnetRxBDs[NUM_RXBDS];
#pragma align EnetRxBDs 8
NBUF EnetTxBDs[NUM_TXBDS];
#pragma align EnetTxBDs 8
/*---------------------------------------------------*/
/* Receive and Transmit Buffers                       */
/*---------------------------------------------------*/
UByte EnetRxBuffer[RX_BUFFER_SIZE * NUM_RXBDS];
#pragma align EnetRxBuffer 32
UByte EnetTxBuffer[TX_BUFFER_SIZE * NUM_TXBDS];
#pragma align EnetTxBuffer 32
```

The previous code defines a *.enetdata* section for the linker file memory configuration. In addition, receive and transmit buffer descriptors and buffers are declared. Note that the buffer descriptors are 8-byte aligned and buffers are 32-byte aligned in memory using the `pragma align` statements.

The actual packet data payload for transmission is defined in the ethernet.c file.

```
/*-----------------------------------------------------*/
/* Data to be transmitted                              */
/*-----------------------------------------------------*/
const UByte packet[] =
{
        0x00, 0xCF, 0x52, 0x82, 0xC3, 0x01, 0x00, 0xCF,
        0x52, 0x82, 0xC3, 0x01, 0x08, 0x00, 0x45, 0x00,
        0x00, 0x3C, 0x2B, 0xE8, 0x00, 0x00, 0x20, 0x01,
        0xA6, 0x1B, 0xA3, 0x0A, 0x41, 0x55, 0xA3, 0x0A,
        0x41, 0x54, 0x08, 0x00, 0x0C, 0x5C, 0x01, 0x00,
        0x40, 0x00, 0x61, 0x62, 0x63, 0x64, 0x65, 0x66,
        0x67, 0x68, 0x69, 0x6A, 0x6B, 0x6C, 0x6D, 0x6E,
        0x6F, 0x70, 0x71, 0x72, 0x73, 0x74, 0x75, 0x76
};
```

The `main()` function is located in ethernet.c file.

```
        *((UWord32 *)(0x10000)) = PASS;
```

The memory address 0x10000 is used throughout the exercise to store PASS or FAIL flags. At the beginning of any exercise the PASS flag is set, and then changed only at the first occurrence of any type of failure. In case of failure, the flag is set to FAIL.

```
        for (i = 0; i < NUM_RXBDS; i++)
        {
                EnetRxBDs[i].status = RX_BD_E;
                EnetRxBDs[i].length = 0;
                EnetRxBDs[i].data = &EnetRxBuffer[i * RX_BUFFER_SIZE];
        }
```

Next, the receive descriptor ring is initialized (Bit 15, or the E bit, is set, indicating that the receive BD is empty; length is changed to 0 and the receive data buffer pointer is initialized).

```
        EnetRxBDs[NUM_RXBDS - 1].status |= RX_BD_W;
```

The wrap bit is set, indicating that the BD is the final one in the receive BD ring, thus completing the ring.

```
        for (i = 0; i < NUM_TXBDS; i++)
        {
                EnetTxBDs[i].status = TX_BD_L | TX_BD_TC;
                EnetTxBDs[i].length = 0;
                EnetTxBDs[i].data = &EnetTxBuffer[i * TX_BUFFER_SIZE];
        }
```

Similarly, the transmit descriptor ring is initialized (Bits 11 and 10, or the L and TC bits, are set, indicating that transmit BD is last in the transmit frame and enabling the transmission of the CRC sequence after the last data byte and transmit data buffer pointer is initialized).

```
        EnetTxBDs[NUM_TXBDS - 1].status |= TX_BD_W;
```

Set the wrap bit, to indicate that BD is the final one in the transmit BD ring, thus completing the ring.

The next block of code configures the FEC controller registers.

```
        Ethernet->PADDR1 = 0x00CF5282;
        Ethernet->PADDR2 = 0xC3018808;
```

PADDRL (PADDR1) contains the lower 32 bits and PADDRH (PADDR2) contains the upper 16 bits of the 48 bit address used in the address recognition process to compare with the destination address. These bits are set to an arbitrary source address value.

```
Ethernet->IADDR[1] = 0x00000000;
Ethernet->IADDR[0] = 0x00000000;
```

Both IADDR1 and IADDR0 contain the upper and lower 32 bits of the 64 bit individual address hash table. This address is used in the address recognition process to check for a match. These registers are not used in this demo, so both are set to 0x0000_0000.

```
Ethernet->GADDR[1] = 0x00000000;
Ethernet->GADDR[0] = 0x00000000;
```

Both GADDR1 and GADDR2 contain the upper and lower 32 bits of the 64 bit hash table, used in the address recognition process for receive frames with a multicast address. These registers are not used in this demo, so both are set to 0x0000_0000.

```
Ethernet->R_BUFF_SIZE = (UWord16)RX_BUFFER_SIZE;
```

R_BUFF_SIZE (RBSZ), receive buffer size register, specifies the maximum size of all receive buffers. This value must be divisible by 16 and its value can be changed in nbuf.h file.

```
Ethernet->R_DES_START = (UWord32)EnetRxBDs;
```

R_DES_START (RDESST), receive descriptor ring start register, holds a pointer to the start of the circular receive BD ring. This memory pointer must be 32-bit word aligned and points to the beginning of EnetRxDBs.

```
Ethernet->X_DES_START = (UWord32)EnetTxBDs;
```

X_DES_START (TDESST), transmit descriptor ring start register, holds a pointer to the start of the circular transmit BD ring. This memory pointer must be 32-bit word aligned and points to the beginning of EnetTxDBs.

```
Ethernet->R_CNTRL =0x05EE0005;
```

R_CNTRL (RCTL), receive control register, is configured for MII mode with internal loopback.

RCTL Receive Control Register ENET_BASE + 0x084

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
			—								MAXFL					
TYPE			R								R/W					
SET	0	0	0	0	0	1	0	1	1	1	1	0	1	1	1	0
	15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
					—						FCE	BFR	PROM	MIIM	DRT	LOOP
TYPE					R								R/W			
SET	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

RCTL controls the operational mode of the receive block and should be written only when EEN = 0, that is, at initialization.

Table 1. RCTL Bit Descriptions

Name	Reset	Description	Settings
— 31–27	0	Reserved. Write to zero for future compatibility.	
MAXFL 26–16	0b101_1110_ 1110 (0x5EE)	**Maximum Frame Length** User-defined maximum frame length, where frame length is measured starting at DA and including the CRC at the end of the frame. Transmit frames longer than MAXFL cause the BABT interrupt. Receive frames longer than MAXFL cause the BABR interrupt to occur and set the LG bit in the end-of-frame BD. The recommended default is the reset value decimal 1518 or decimal 1522 if VLAN tags are supported.	0x5EE (or 1518 decimal)
— 15–6	0	Reserved. Write to zero for future compatibility.	
FCE 5	0	**Flow Control Enable** Enables the receiver to detect pause frames. When a pause frame is detected, the transmitter stops transmitting data frames for a specified duration.	0 Normal operation.
BFR 4	0	**Broadcast Frame Reject** Causes frames with a destination address (DA) = FFFF_FFFF_FFFF to be rejected unless the PROM bit is set. If both BFR and PROM = 1, frames with broadcast DA are accepted and the miss (M) bit is set in the receive BD.	0 Disabled
PROM 3	0	**Promiscuous Mode** All frames are accepted, regardless of address matching.	0 Disabled
MIIM 2	0	**External Interface Mode** Selects between 7Wire Interface mode and MII mode, which applies to both the transmit and receive blocks.	1 MII operating mode.
DRT 1	0	**Disable Receive on Transmit** Disables reception of frames while transmitting, which is normally used for half duplex mode.	0 Receive path operates independently of transmit (use for full duplex or to monitor transmit activity in half duplex mode).
LOOP 0	1	**Internal Loopback** Causes transmitted frames to be looped back internal to the device, and the transmit output signals are not asserted. The system clock is substituted for the TXCLK when LOOP is asserted. DRT must be cleared when LOOP is asserted.	1 Internal loopback.

```
        Ethernet->X_CNTRL = 0x0004;
```

X_CNTRL (TCTL), transmit control register is configured for full-duplex mode.

TCTL Transmit Control Register ENET_BASE + 0x0C4

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	—															
TYPE	R															
SET	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	—											RFCP	TFCP	FDEN	HBC	GTS
TYPE	R												R/W			
SET	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

Table 2. Transmit Control Register Bit Descriptions

Name	Reset	Description	Settings
— 31–5	0	Reserved. Write to zero for future compatibility.	
RFCP 4	0	**RFC Pause** Set when a full duplex flow control pause frame is received. The transmitter pauses for the duration defined in the pause frame This bit automatically clears when the pause duration is complete.	0 Normal operation.
TFCP 3	0	**TFC Pause** When this bit is set, the MAC stops transmitting data frames after the current transmission completes. The GRA interrupt in the IEVENT register is asserted. The MAC transmits a MAC Control PAUSE frame. Next, the MAC clears the TFCP bit and resumes transmitting data frames. If the transmitter is paused due to user assertion of GTS or reception of a PAUSE frame, the MAC can still transmit a MAC Control PAUSE frame.	0 Normal operation.
FDEN 2	0	**Full Duplex Enable** If set, frames are transmitted independent of carrier sense and collision inputs. This bit should be modified only when ECTL[EEN] is cleared.	1 Enable full duplex.
HBC 1	0	**Heartbeat Control** If set, the heartbeat check is performed at the end of transmission and the HB bit in the status register is set if the collision input does not assert within the heartbeat window. This bit should be modified only when ECTL[EEN] is cleared.	0 Normal operation.
GTS 0	0	**Graceful Transmit Stop** When this bit is set, the MAC stops transmission after the frame that is being transmitted is complete, and the GRA interrupt in the IEVENT register is asserted. If no frames are being transmitted, the GRA interrupt is asserted immediately. When transmission completes, the GTS bit is cleared to initiate a restart. The next frame in the transmit FIFO is then transmitted. If an early collision occurs during transmission when GTS = 1, transmission stops after the collision. The frame is transmitted again when GTS is cleared. If old frames are in the transmit FIFO, they are transmitted when GTS is reasserted. To avoid this, clear ECTL[EEN] after the GRA interrupt.	0 Normal operation.

```
pNbuf = EnetTxBDs;
for (i = 0; i < 64; i++)
{
         pNbuf->data[i] = packet[i];
}
```

In the code above, the packet payload is loaded into buffer through the transmit buffer descriptor.

```
pNbuf->length = 64;
```

The length of the same buffer descriptor is updated to reflect the length of the packet.

```
Ethernet->ECNTRL = 0x00000001; // Clear the ECR
Ethernet->ECNTRL = 0x00000002;
```

ECNTRL (ECTL), ethernet control register, is first cleared and bit 0 (RESET bit) is set in order to reset the FEC. Next the ECTL is programmed to release reset and set the EEN bit to enable the FEC.

ECTL Ethernet Control Register ENET_BASE + 0x024

Bit	**31**	**30**	**29**	**28**	**27**	**26**	**25**	**24**	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**
	TAG3	TAG2	TAG1	TAG0	—	TMD	—									
TYPE	R/W				R	R/W	R									
SET	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	—														EEN	RESET
TYPE	R														R/W	
SET	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

Table 3. Ethernet Control Register Bit Descriptions

Name	Reset	Description	Settings
TAG[3–0] 31–28	0b1111	**Tags 3–0** Programs and reads the TBus tag bits. This field, which is used for debug/test only, is composed of two separate 4-bit registers, tags-in and tags-out. A write from the IPBus to this field is written to tags-in. During a write cycle to any FEC register other than ECTL, the tags-in value is driven onto the TBus data bus tag field. During a read cycle, the TBus tag field bits are latched and saved in the tags-out register. When the ECTL register is read from the IPBus interface, the value from tags-out appears in the TAG[3–0] field.	0
— 27	0	Reserved. Write to zero for future compatibility.	
TMD 26	0	**Test Mode**	0
— 25–2	0	Reserved. Write to zero for future compatibility.	

Name	Reset	Description	Settings
EEN 1	0	**Ethernet Enable** Enables/disables the FEC. When this bit is set, the Ethernet can receive and transmit data. When this bit is cleared, reception immediately stops, and transmission stops after a bad CRC is appended to any frame being transmitted. The BD(s) for an aborted transmit frame are not updated following deassertion of EEN. When EEN is deasserted, the DMA controller, BD, and FIFO control logic is reset, including BD and FIFO pointers. When software writes a value of 1 to ECTL[RESET] or an AHB bus error is detected, the hardware clears EEN. The procedure for halting the FEC is described in **Section 11.4.4.3**, *Complete Halt of the Ethernet MAC,* in the *MC711x Reference Manual.*	1 Ethernet enabled.
RESET 0	0	**Ethernet Controller Reset** When this bit is set, the equivalent of a hardware reset is performed but it is local to the FEC. EEN is cleared, and all other FEC registers take their reset values. Also, any transmission/reception in progress is abruptly aborted. Hardware automatically clears this bit during the reset sequence, which requires approximately eight clock cycles after RESET is written with a 1. **Note:** Before using the RESET bit to reset the FEC, shut down the FEC as described in **Section 11.4.4.3**, *Complete Halt of the Ethernet MAC,* in the *MC711xReference Manual*.	0 Normal operation.

```
        for (i = 0; i < NUM_RXBDS; i++)
        {
                EnetRxBDs[i].status = RX_BD_E;
                EnetRxBDs[i].length = 0;
                EnetRxBDs[i].data = &EnetRxBuffer[i * RX_BUFFER_SIZE];
        }
        EnetRxBDs[NUM_RXBDS - 1].status |= RX_BD_W;
```

Clear and setup the receive buffers (Set the wrap bit on the last BD in the ring).

```
        Ethernet->R_DES_ACTIVE = MSC711x_FEC_RDAR_R_DES_ACTIVE;
```

R_DES_ACTIVE (RDA), receive descriptor active register (RDA bit is set) to activate receive descriptor.

RDA Receive Descriptor Active Register ENET_BASE + 0x010

Bit	**31**	**30**	**29**	**28**	**27**	**26**	**25**	**24**	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**
	—							RDA	—							
TYPE	R							R/W	R							
SET	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
	15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	—															
TYPE	R															
SET	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

RDA is a programmable command register. The user writes to this register to indicate that the receive descriptor ring has been updated; that is, the driver has produced empty receive buffers. When the register is written, the RDA bit is set, independently of the data written by the user. When RDA is set and the ECTL[EEN] bit is also set, the FEC polls the receive descriptor ring and processes the receive frames. When the FEC polls a receive descriptor whose empty (E) bit is not set, the FEC clears RDA and ceases polling the receive descriptor rings until the RDA bit is set again, indicating that the driver has produced an empty receive buffers. This register is cleared not only at reset but also at the clearing of the ECTL[EEN] bit.

Table 4. RDA Bit Descriptions

Name	Reset	Description	Settings
— 31–25	0	Reserved. Write to zero for future compatibility.	
RDA 24	0	**Receive Descriptor Active** Set to one when this register is written, regardless of the value written. Cleared by the FEC device when no additional ready descriptors remain in the receive ring.	1 - Receive Descriptor Active
— 23–0	0	Reserved. Write to zero for future compatibility.	

```
        pNbuf->status |= TX_BD_R;
```

Change the R bit of transmit buffer descriptor to 1, to indicate that the packet is ready to send.

```
        Ethernet->X_DES_ACTIVE = MSC711x_FEC_TDAR_X_DES_ACTIVE;
```

X_DES_ACTIVE (TDA), transmit descriptor active register (TDA bit is set) to activate transmit descriptor.

TDA Transmit Descriptor Active Register 0x014

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	—							TDA	—							
TYPE	R							R/W	R							
SET	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	—															
TYPE	R															
SET	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

TDA is a programmable command register. The register is written by the user to indicate that the transmit descriptor ring has been updated; that is, the driver has produced transmit buffers. When the register is written, the TDA bit is set, independently of the data actually written by the user. When TDA is set and ECTL[EEN] is also set, the FEC polls the transmit descriptor ring and processes transmit frames. When the FEC polls a transmit descriptor whose ready (R) bit is not set, the FEC clears TDA and ceases polling transmit descriptor rings until the user sets TDA again to indicate that more descriptors are in the transmit descriptor ring. This register is cleared not only at reset but also at the clearing of the ECTL[EEN] bit.

Table 5. TDA Bit Descriptions

Name	Reset	Description	Settings
— 31–25	0	Reserved. Write to zero for future compatibility.	
TDA 24	0	**Transmit Descriptor Active** Set to one when this register is written, regardless of the value written. Cleared by the FEC device when no additional ready descriptors remain in the transmit ring.	1. Transmit Descriptor Active
— 23–0	0	Reserved. Write to zero for future compatibility.	

```
        for (i=0; i < 5000; i++)
        {
                if (Ethernet->IEVENT & MSC711x_FEC_EIR_RXF)
                {
                        break;
                }
        }
        if (i == 5000)
        {
                // Timed-out
                *((UWord32 *)(0x10000)) = FAIL;
                asm(" debug");
                return 0;
        }
```

Arbitrary delay for allowing packets to propagate from transmit to receive interface. If the delay is greater then 5000, set the FAIL flag/condition due to the time out.

```
        for (i = 0; i < 64; i++)
        {
                if (EnetTxBDs[0].data[i] != EnetRxBDs[0].data[i])
                {
                        *((UWord32 *)(0x10000)) = FAIL;
                        asm(" debug");
                        return 0;
                }
        }

// Ethernet Test passed
        asm(" debug");
```

Check and ensure that the data in receive buffers is the same as in transmitted buffers. If the data is not the same set the FAIL flag.

4 MII Internal Loop with Interrupts

In this second example, the interrupts are introduced as a means of interacting with the FEC. This example builds on the knowledge and code in the first example. The changes in the ethernet.c file are presented first, focusing mostly on `main()` and then the functions used in the msc711x_int.c file, which has interrupt related functionality, are given.

4.1 ethernet.c and main() differences

```
#include <prototype.h>
#include "include\msc711x_int.h"
```

Two new files are included:

```
volatile int interrupted=0;
volatile int interruptedNoOfTimes_RX=0;
volatile int interruptedNoOfTimes_TX=0;
volatile int interruptedNoOfTimes_EIR_HBERR=0;
volatile int interruptedNoOfTimes_EIR_BABR=0;
volatile int interruptedNoOfTimes_EIR_BABT=0;
volatile int interruptedNoOfTimes_EIR_GRA=0;
volatile int interruptedNoOfTimes_EIR_TXB=0;
volatile int interruptedNoOfTimes_EIR_RXB=0;
volatile int interruptedNoOfTimes_EIR_MII=0;
volatile int interruptedNoOfTimes_EIR_EBERR=0;
volatile int interruptedNoOfTimes_EIR_LC=0;
volatile int interruptedNoOfTimes_EIR_RL=0;
volatile int interruptedNoOfTimes_EIR_UN=0;
```

After packet definition these variables are defined and initialized to 0. These variables can be used to track different counters associated with receive and transmit operations of FEC.

Right before `main()`, interrupt handing functions are defined. The first function, `EnetTxFRxFIntHandler()` detects and counts how many times receive and transmit interrupts are called. This proves very useful for debugging.

```
void EnetTxFRxFIntHandler(void)
{
        enet_map_t* Ethernet= (enet_map_t*)ENET_BASE;
```

```
Set the memory map address and base for accessing the FEC registers.
```

```
        asm(" di");
```

Disable interrupts.

```
        if(Ethernet->IEVENT & MSC711x_FEC_EIR_RXF)
        {
                interruptedNoOfTimes_RX++;
                interrupted=1;
        }
```

Use the IEVENT[RFINT] bit, to detect if the receive frame interrupt has occurred.

IEVENT Interrupt Event Register ENET_BASE + 0x004

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	HBERR	BABR	BABT	GRA	TFINT	TXB	RFINT	RXB	MII	—	LC	CRL	TFU	ROV	—	
TYPE	R/W									R	R/W				R	R
RESET	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	—															
TYPE	R															
RESET	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 6. IEVENT Bit Descriptions

Name	Reset	Description	Settings
HBERR 31	0	**Heartbeat Error** If the CTL[HBC] bit is set and the FEC does not detect a collision within the heartbeat window, a heartbeat error occurs. The FEC closes the buffer and generates the HBERR interrupt if it is enabled. The heartbeat condition is checked only in Half Duplex mode.	0 Heartbeat error disabled. 1 Heartbeat error enabled.
BABR 30	0	**Babbling Receive Error** Indicates a frame received with a length in excess of RCTL[MAXFL], which specifies the maximum frame length, in bytes.	0 No interrupt. 1 Babbling receive error interrupt.

Table 6. IEVENT Bit Descriptions (continued)

Name	Reset	Description	Settings
BABT 29	0	**Babbling Transmit Error** Indicates a frame transmitted with a length in excess of RCTL[MAXFL]. This condition is usually caused when a frame that is too long is placed into the transmit data buffer(s). Truncation does not occur.	0 No interrupt. 1 Babbling transmit error interrupt.
GRA 28	0	**Graceful Stop Complete** Graceful stop means that the transmitter is put into a pause state after completion of the frame being transmitted.This interrupt is asserted for one of three reasons. 1. A graceful stop initiated by setting the TCTL[GTS] bit is now complete. 2. A graceful stop initiated by setting the TCTL[FCP] bit is now complete. 3. A graceful stop initiated by the reception of a valid full duplex flow control pause frame is now complete. Refer to **Section 18.4.7**, *Full Duplex Flow Control*, in the *MSC711x Reference Manual*.	0 No interrupt. 1 Graceful stop complete interrupt.
TFINT 27	0	**Transmit Frame Interrupt** Indicates that a frame has been transmitted and that the last corresponding buffer descriptor (BD) has been updated. This interrupt is generated when the transmit block generates status for the just completed frame.	0 No interrupt. 1 Transmit frame completed interrupt.
TXB 26	0	**Transmit Buffer Interrupt** Indicates that a transmit BD has been updated. This interrupt is generated when a DMA transfer of a transmit buffer is complete.	0 No interrupt. 1 Transmit buffer interrupt.
RFINT 25	0	**Receive Frame Interrupt** Indicates that a frame has been received and that the last corresponding BD has been updated. This bit is set after the last receive buffer in a frame has been transferred via DMA.	0 No interrupt. 1 Receive frame interrupt.
RXB 24	0	**Receive Buffer Interrupt** Indicates that a receive BD has been updated that was not the last in the frame. This bit is set upon completion of a DMA transfer of a receive buffer that is not the last in the frame.	0 No interrupt. 1 Receive buffer interrupt.
MII 23	0	**MII Interrupt** Indicates that the MII has completed the requested data transfer. This bit is set if the transceiver register read/write operation controlled by the MIIDATA and MIISPEED registers is complete.	0 No interrupt. 1 MII interrupt.
— 22	0	Reserved. Write to zero for future compatibility.	

Table 6. IEVENT Bit Descriptions (continued)

Name	Reset	Description	Settings
LC 21	0	**Late Collision** Indicates a collision beyond the collision window (slot time) in half duplex mode. The frame is truncated with a bad CRC. and the remainder of the frame is discarded. In Full Duplex mode, the collision input is ignored.	0 No interrupt. 1 Late collision interrupt.
CRL 20	0	**Collision Retry Limit** Indicates a collision on each of 16 successive attempts to transmit the frame. The frame is discarded without being transmitted, and transmission of the next frame commences. This interrupt can occur only in Half Duplex mode.	0 No interrupt. 1 Collision retry limit interrupt.
TFU 19	0	**Transmit FIFO Underrun** Indicates that the transmit FIFO emptied before the complete frame was transmitted. A bad CRC is appended to the frame fragment, and the remainder of the frame is discarded.	0 No interrupt. 1 Transmit FIFO underrun interrupt.
ROV 18	0	**Receiver Overrun** Indicates that the receiver is full and has dropped frame data during reception. The OV bit in the corresponding RxBD is set, indicating that the frame should be discarded.	0 No interrupt. 1 Receiver overrun interrupt.
— 17–0	0	Reserved. Write to zero for future compatibility.	

```
        if(Ethernet->IEVENT & MSC711x_FEC_EIR_TXF)
        {
                interruptedNoOfTimes_TX++;
        }
```

Similarly, as for the receive side, the IEVENT register is used to determine if the transmit frame interrupt occurred by examining the IEVENT[TFINT] bit.

```
        Ethernet->IEVENT=0x0A000000;
```

Clear both transmit and receive frame interrupt flags by writing one.

```
        asm(" ei");
```

Enable interrupts

```
        return;
}
```

Function `EnetTxFRxFVector()` is used as a interrupt vector, which calls the `EnetTxFRxFIntHandler()` function, which handles the interrupt.

```
void EnetTxFRxFVector(void)
{
        #pragma interrupt EnetTxFRxFVector
        EnetTxFRxFIntHandler();
}
```

Function `EnetSummaryIntHandler` is used to keep count of different FEC errors and/or conditions which could occur during receive and transmit operations.

```
void EnetSummaryIntHandler(void)
{
        #pragma interrupt EnetSummaryIntHandler
        enet_map_t* Ethernet= (enet_map_t*)ENET_BASE;
        asm(" di");
```

Disable interrupts.

```
        if (Ethernet->IEVENT & MSC711x_FEC_EIR_HBERR) interruptedNoOfTimes_EIR_HBERR++;
        if (Ethernet->IEVENT & MSC711x_FEC_EIR_BABR) interruptedNoOfTimes_EIR_BABR++;
        if (Ethernet->IEVENT & MSC711x_FEC_EIR_BABT) interruptedNoOfTimes_EIR_BABT++;
        if (Ethernet->IEVENT & MSC711x_FEC_EIR_GRA) interruptedNoOfTimes_EIR_GRA++;
        if (Ethernet->IEVENT & MSC711x_FEC_EIR_TXB) interruptedNoOfTimes_EIR_TXB++;
        if (Ethernet->IEVENT & MSC711x_FEC_EIR_RXB) interruptedNoOfTimes_EIR_RXB++;
        if (Ethernet->IEVENT & MSC711x_FEC_EIR_MII) interruptedNoOfTimes_EIR_MII++;
        if (Ethernet->IEVENT & MSC711x_FEC_EIR_EBERR) interruptedNoOfTimes_EIR_EBERR++;
        if (Ethernet->IEVENT & MSC711x_FEC_EIR_LC) interruptedNoOfTimes_EIR_LC++;
        if (Ethernet->IEVENT & MSC711x_FEC_EIR_RL) interruptedNoOfTimes_EIR_RL++;
        if (Ethernet->IEVENT & MSC711x_FEC_EIR_UN) interruptedNoOfTimes_EIR_UN++;
```

IEVENT register bits are checked for the following events:

1. HBEER—Heartbeat Error is set if CTL[HBC] bit is set and FEC does not detects a collision within the heartbeat window.
2. BABR—Babbling Receive Error indicates that maximum frame length has been exceeded.
3. BABT—Babbling Transmit Error indicates that the frame which is too long has been placed into the transmit data buffers.
4. GRA—Graceful Stop Complete indicates that the transmitter is put into a pause state after completion of the frame being transmitted.
5. TXB—Transmit Buffer Interrupt is set when a DMA transfer of a transmit buffer is complete.
6. RXB—Receive Buffer Interrupt is set when a DMA transfer of a receive buffer is complete.
7. MII—MII Interrupt indicates that the MII has completed the requested data transfer.
8. EBERR—is currently reserved.
9. LC—Late Collision indicates that a collision beyond the collision window in half duplex mode has occurred.
10. RL—Collision Retry Limit indicates that a collision on each of 16 successive attempts to transmit the frame has occurred.
11. UN—Transmit FIFO Underrun indicates that the transmit FIFO emptied before the complete frame was transmitted.

```
        Ethernet->IEVENT=0xf5ffffff;
```

Clear all interrupt events in the IEVENT register by writing one to each bit. Note that IEVENT[TFINT] and IEVENT[RFINT] are not cleared as these two events are handled in the `EnetTxFRxFIntHandler()` function.

```
        asm(" ei");
```

Enable interrupts.

```
        return;
```

Function `EnetSummaryVector()` is used as an interrupt vector to call the `EnetSummaryIntHandler()` function, which handles the interrupt.

```
void EnetSummaryVector(void)
{
        #pragma interrupt EnetSummaryVector
        EnetSummaryIntHandler();
}
```

Function `InitInterrupts()` is used to initialize interrupts and is called in the beginning of the `main()`. This function enables all interrupt priority levels in the status registers first, sets up the VBA vector table, and installs the NMI Branch and Auto-Vector Branch table handler. Once these operations are complete the `IntEnetInterrput()` configures two interrupt vectors, which were defined above: `EnetTxFRxFVector()` (pointing to the `EnetTxFRxFVectorHandler()`) and `EnetSummaryVector()` (pointing to the `EnetSummaryIntHandler()`). The interrupt priority levels are also set for each interrupt that is going to be handled by the `EnetTxFRxFVectorHandler()`and the `EnetSummaryIntHandler()`.

```
void InitEnetInterrupt(void)
{
        InitInterrupts();

        CopyChInt(INT_ENTRXF, EnetTxFRxFVector);
        SetInterruptPriorityLevel(INT_ENTRXF, 5);
        SetInterruptPriorityLevel(INT_ENTTXF, 5);

        CopyChInt(INT_ENTSMRY, EnetSummaryVector);
        SetInterruptPriorityLevel(INT_ENTSMRY, 5);
        ei();
}
```

Now that the interrupt functions are introduced, their usage in the `main()` and new application code flow is covered. Since this exercise builds on the first one and is very similar, only the differences are pointed out.

The exercises starts with the variable initialization followed by the `InitEnetInterrput()` function, which initializes the interrupts. Following interrupts initialization, the receive and transmit buffers and buffer descriptors are initialized and FEC configuration registers are set up. All the data initialization and FEC configuration are the same as in the previous exercise, up to this point.

```
        Ethernet->ECNTRL = 0x00000001; // Clear the ECR
        Ethernet->IMASK  = 0x0f000000;
        Ethernet->IEVENT = 0xffffffff;
        Ethernet->ECNTRL = 0x00000002;
```

As in previous exercise, the FEC is prepared for operation by first being reset with the FCNTRL[RESET] bit and then enabled with FCNTRL[EEN] bit. The new addition is between FEC reset and FEC enable where the IMASK is configured to 0x0F00_0000 which enables the following interrupts:

1. Transmit Frame Interrupt (TFIEN)

2. Transmit Buffer Interrupt (TBIEN)
3. Receive Frame Interrupt (RFIEN)
4. Receive Buffer Interrupt (RBIEN)

IMASK Interrupt Enable Register ENET_BASE + 0x008

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	HBEEN	BREN	BTEN	GRAEN	TFIEN	TBIEN	RFIEN	RBIEN	MIIEN	—	LCEN	CRLEN	TFUEN	ROVEN	—	
TYPE	R/W									R	R/W				R	
SET	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0
	15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	—				TFAC	—	RFAC	—								
TYPE	R				R/W	R	R/W	R								
SET	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 7. IMASK Bit Descriptions

Name	Reset	Description	Settings
HBEEN 31	0	**Heartbeat Error Enable** .	0 Not enabled.
BREN 30	0	**Babbling Receive Interrupt Enable** .	0 Not enabled.
BTEN 29	0	**Babbling Transmitter Interrupt Enable**	0 Not enabled.
GRAEN 28	0	**Graceful Stop Interrupt Enable**	0 Not enabled.
TFIEN 27	0	**Transmit Frame Interrupt Enable**	1 Transmit frame interrupt enabled.
TBIEN 26	0	**Transmit Buffer Interrupt Enable**	1 Transmit buffer interrupt enabled.
RFIEN 25	0	**Receive Frame Interrupt Enable**	1 Receive frame interrupt enabled.
RBIEN 24	0	**Receive Buffer Interrupt Enable**	1 Receive buffer interrupt enabled.
MIIEN 23	0	**MII Interrupt Enable**	0 Not enabled.
— 22	0	Reserved. Write to zero for future compatibility.	
LCEN 21	0	**Late Collision Enable**	0 Not enabled.
CRLEN 20	0	**Collision Retry Limit Enable**	0 Not enabled.

Table 7. IMASK Bit Descriptions (continued)

Name	Reset	Description	Settings
TFUEN 19	0	**Transmit FIFO Underrun Enable**	0 Not enabled.
ROVEN 18		**Receiver Overrun Enable**	0 Not enabled.
— 17–12	0	Reserved. Write to zero for future compatibility.	
TFAC 11	0	**Transmit Frame Interrupt Automatic Clear** Enables automatic clearing of the transmit frame interrupt after one cycle of assertion. TFAC allows an interrupt to be generated for several transmit frames using the event port and a timer configured to detect TFINT rising edges.	0 No automatic clear of TFINT.
— 10	0	Reserved. Write to zero for future compatibility.	
RFAC 9	0	**Receive Frame Interrupt Automatic Clear** Enables automatic clearing of the receive frame interrupt after one cycle of assertion. RFAC allows an interrupt to be generated for several receive frames using the event port and a timer configured to detect RFINT rising edges.	0 No automatic clear of RFINT.
— 8–0	0	Reserved. Write to zero for future compatibility.	

For completeness, IEVENT is set to pair with the IMASK register. The value of 0xFFFF_FFFF is written to complete the interrupt enable functionality. Write one to each of the interrupt bits to clear the interrupt event condition for each bit-interrupt pair.

Once the FEC is enabled the data is sent from the transmit to receive side. As before, wait for the data to propagate.Since interrupts are used, the compare method is different, the old method of polling the IEVENT register in `main()`, is replaced by simply checking the interrupted flag. This flag is set in the Ethernet receive interrupt handler function.

```
for (i=0; i < 5000; i++)
{
        if (interrupted)
        {
                break;
        }
}
```

Once the data is received, the context of the receive packets is checked and if everything is correct, the exercise ends with the PASS flag.

4.2 msc711x_int.c

The msc711x_int.c file contains all interrupt related code which is added to this exercise. This is a brief summary of each function used:

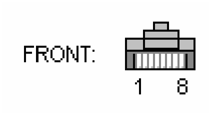
1. `SetVBA` —Sets the location of the VBA table.
2. `CopyChInt` and `CopyCoreInt` —Used for installing core and application interrupt vector location and parameters.
3. `DisableAllMaskableInterrputs` —Disables all 16 maskable interrupts
4. `SetInterrputPriorityLevel` —Set particular interrupts priority level, by using the IPLR register.
5. `Auto_handler`, `InstallAutoHandler` and `AutoIsr` —Not used in this exercise but included for completeness of the msc711x_int.c compilation of routines.
6. `UnhandledNMI` - Sets up debug() trap just in case unhandled NMI occurs.
7. `InstallNmiHandler`, `NmiIsr` and `ClearNMI` — Handle the installation and operation of the Non-Makable-Interrupts (NMI).
8. `IntInterrputs` —Called at the beginning of the `main()` for initialization of the interrupts used in this exercise.

5 MII External Loop with Interrupts

In this third example, the date is sent externally from transmit to receive circuitry through PHY and Ethernet loopback cable. This exercise uses the MSC711xADS with Davicom (DM9161E) PHY. Depending on the 711x board and/or PHY combination, some PHY initialization changes might be necessary. This example builds on the knowledge and code of the first and second example. All additional code changes in ethernet.c, which are necessary to get the FEC to send the Ethernet packets externally through PHY, are reviewed.

5.1 Ethernet Loopback Cable

For this example, the internal software loopback capability is not used to exercise the Ethernet PHYs. An Ethernet loopback cable is necessary to loopback the data. To create an Ethernet loopback cable, first start with a standard twisted-pair Ethernet cable with RJ45 connectors.



The pin-out for the RJ45 connector is as shown in Table 5-8:

Table 5-8. Pin-out to Wire Color for RJ45 Connector

Pin	Wire Color
1	White-Orange
2	Orange
3	White-Green
4	Blue
5	White-Blue
6	Green
7	White-Brown
8	Brown

Ethernet loopback cable assembly:

1. Cut the end off the Ethernet cable approximately 6 inches from the RJ45 connector.
2. Strip wires 1 (white-orange), 2 (orange), 3(white-green) and 6 (green).
3. Twist and solder wires 1 and 3 together (white-orange to white-green).
4. Twist and solder wires 2 and 6 together (orange to green).
5. To protect from shorting the exposed wires, insulate them with electrical tape.
6. Leave all other wires as they are.

5.2 Implementation

As in the second example, this example starts with variable initialization, but one variable is added, which is used to keep track of MIIDATA and MIISPEED register status.

```
volatile int interrupted_EIR_MII=0;
```

This variable is set to 1 in the `EnetSummaryIntHandler()` function, along with the old interruptedNoOfTimes_EIR_MII counter.

```
        if (Ethernet->IEVENT & MSC711x_FEC_EIR_MII)
        {
                interruptedNoOfTimes_EIR_MII++;
                interrupted_EIR_MII = 1;
        }
```

The following functions are added for handling the PHY interface.

```
void my_delay(int n)
{
        volatile int nn = n;
        while (nn > 0) {
                nn -= 1;
        }
}
```

Function `my_delay()` was introduced because of the need to make variable delays for different PHYs. A PHY might require differed delay values during PHY initialization.

```
unsigned short ReadPhy(long reg_addr)
{
        unsigned short temp;
        enet_map_t* Ethernet= (enet_map_t*)ENET_BASE;

        Ethernet->MII_SPEED = 0x50;
        Ethernet->MII_DATA = 0x60020000 | ( reg_addr<<18);

        while(!interrupted_EIR_MII);
        interrupted_EIR_MII=0;

        temp = (unsigned short)(0x0000FFFF & Ethernet->MII_DATA);

        Ethernet->MII_SPEED = 0;
        Ethernet->MII_DATA  = 0;

        return temp;
}
```

`ReadPhy` function is called from the `InitPhy` function during PHY initialization for polling PHY.

```
void WritePhy(long reg_addr,unsigned long data)
{
        enet_map_t* Ethernet= (enet_map_t*)ENET_BASE;

        Ethernet->MII_SPEED = 0x50;
        Ethernet->MII_DATA = 0x50020000 | ( reg_addr<<18) | data;

        while(!interrupted_EIR_MII);
        interrupted_EIR_MII=0;

        Ethernet->MII_SPEED = 0;
        Ethernet->MII_DATA  = 0;

        return;
}
```

Similar to `ReadPhy`, the `WritePhy` function is called from InitPhy during PHY initialization. This function writes configuration commands, such as reset and mode setup, to the PHY.

```
void InitPhy()
{
        unsigned short i=0;

        WritePhy(MII_CR, MII_CR_RESET);
        while((ReadPhy(MII_CR) & MII_CR_RESET))
        {
                i++;
        }
        WritePhy(MII_CR,0x2100);
}
```

Function `InitPhy` is called once in the beginning of `main()` for PHY initialization. Upon completion of this function PHY is initialized (MII_CR is configured to 0x2100) and after a certain delay PHY is ready.

```
void InitGPIO(){
        gpio_map_t *pstGPIO;
        pstGPIO = (gpio_map_t *)(GPIO_BASE);

        pstGPIO->gp[0].GP_CTL = 0x3FF80000;
        pstGPIO->gp[0].GP_DR = 0;
        pstGPIO->gp[0].GP_DDR = 0;

        pstGPIO->gp[3].GP_CTL = 0x0000007F;
        pstGPIO->gp[3].GP_DR = 0;
        pstGPIO->gp[3].GP_DDR = 0;
}
```

Function `initGPIO()` initializes and configures the general purpose input and output pins for Ethernet functionality. The values 0x3FF8_0000 for port A and 0x0000_007F for port D imply following GPIO configuration:

```
*   PHY_TXER      (TXD4)           PA-28
*   ETH_TXD3      -                PA-27
*   ETH_TXD2      -                PD-4
*   ETH_TXD1      -                PA-19
*   ETH_TXD0      -                PA-20
*   ETH_TXEN      -                PA-24
*   PHY_TXCLK     (ISOLATE)        PA-23
*   ETH_RXER      (RXD4/RPTR)      PA-26
*   ETH_RXD3      (PHYAD3)         PA-29
*   ETH_RXD2      (PHYAD2)         PD-6
*   ETH_RXD1      (PHYAD1)         PA-21
*   ETH_RXD0      (PHYAD0)         PA-22
*   ETH_RXDV      (TESTMODE)       PA-25
*   SL_ETH_EN     (RXEN)           BCSR
*   ETH_RXCLK     (SCRM/10BTSER)   PD-5
*   MDC           MDC              PD-2
*   MDIO          MDIO             PD-3
*   nIRQ2         MDINTR           8272
*   ETH_LINK      CBLSTS/LINKSTS   10/100M ETH PLUG
*   ETH_COL       RMII             PD-0
*   ETH_CRS       CRS/PHYAD4       PD-1
*   nHRSET_SL     RESET            JTAG
*
```

```
void CheckData(int index)
{
        int j;
        for (j = 0; j < 64; j++)
        {
                if (EnetTxBDs[index].data[j] != EnetRxBDs[index].data[j])
                {
                        *((UWord32 *)(0x10000)) = FAIL;
                        asm(" debug");
                }
        }
}
```

Once the data is transmitted and received, the function `CheckData` is called to verify that the transmitted data matches the received data. In case of a mismatch, the FAIL flag is set and execution is halted at the debug statement.

Function `main()` is very similar to the previous example, with small additions for infinite Ethernet Rx and Tx loopback mode.

```
        Word32 statInfLoop=0;
```

This variable can be used to count the number of complete loop iterations.

```
        InitGPIO();
```

Next initialize and configure the general purpose IO pins for Ethernet functionality.

```
        InitPhy();
```

As soon as GPIO pins are configured, the Ethernet PHY can be initialized, and that is performed right after this function:

```
        my_delay(5000);
```

This delay is necessary for the PHY being used. This delay could be reduced or eliminated altogether depending on the PHY.

```
        for (i = 0; i < NUM_TXBDS; i++)
        {
                EnetTxBDs[i].status = TX_BD_L | TX_BD_TC | TX_BD_R;
                EnetTxBDs[i].length = 64;
                EnetTxBDs[i].data = &EnetTxBuffer[i * TX_BUFFER_SIZE];
                for (j = 0; j < 64; j++)
                {
                        EnetTxBDs[i].data[j] = packet[j];
                }
        EnetTxBDs[NUM_TXBDS - 1].status |= TX_BD_W;
```

The code above initializes the transmit buffer descriptor rings and loads the data to be transmitted.

```
        Ethernet->R_CNTRL = (RX_BUFFER_SIZE << 16) | 0x4;
//      R_CNTRL  Max frame length  = RX_BUFFER_SIZE
//                                 Flow Control Enable        = 0
//                                 Broadcast frame reject     = 0
//                                 Promiscuous mode           = 0
//                                 MII mode                   = 1
//                                 Disable Rx or Tx           = 0
//                                 Internal loopback          = 0
```

Since we are no longer using the internal loopback option, the internal loopback bit is cleared to 0 in the R_CNTRL register.

```
        Ethernet->CFGR = 0x00;
```

Initialize the MIIGSK configuration register (MIIGSKCFG) with 0x00 for MII-bridge and pass thorough mode. For debugging purposes, value 0x11 can be written to enable internal loopback at the MII-Bridge level.

```
        Ethernet->ENR = 0x1;
```

Ethernet transmission and reception of frames is enabled with ENR register.

```
        while (Ethernet->ENR != 0x3);
```

Next is the `main()` loop. First, prepare the receive buffer by enabling polling and transmit buffers by indicating that they are ready to send data. Then, wait to see if the frame is received in the receive complete/done interrupt. Next, as before, the time out condition is checked, followed by re-initialization of the receive and transmit buffer descriptor rings.

```
         while(1)
         {
                  for (i = 0; i < NUM_TXBDS; i++)
                  {
                           Ethernet->R_DES_ACTIVE = MSC711x_FEC_RDAR_R_DES_ACTIVE;
                           Ethernet->X_DES_ACTIVE = MSC711x_FEC_TDAR_X_DES_ACTIVE;
                           for (d=0; d < 5000; d++)
                           {
                                    if (interrupted)
                                    {
                                             interrupted = 0;
                                             break;
                                    }
                           }
                           if (d == 5000)
                           {
                                    *((UWord32 *)(0x10000)) = FAIL;
                                    asm(" debug");
                                    return 0;
                           }

                           if (EnetRxBDs[i].status & RX_BD_E)
                           {
                                    *((UWord32 *)(0x10000)) = FAIL;
                                    asm(" debug");
                                    return 0;
                           }
                           CheckData(i);

                           EnetRxBDs[i].status = RX_BD_E;
                           EnetRxBDs[i].length = 0;

                           if (i == (NUM_RXBDS - 1))  {
                                    EnetRxBDs[NUM_RXBDS - 1].status |= RX_BD_W;
                           }

                           EnetTxBDs[i].status = TX_BD_L | TX_BD_TC | TX_BD_R;
                           if (i == (NUM_TXBDS - 1))
                           {
                                    EnetTxBDs[NUM_TXBDS - 1].status |= TX_BD_W;
                           }
                  }
                  statInfLoop++;
         }
}
```

6 Revision History

Table 9 provides a revision history for this application note.

Table 9. Document Revision History

Rev. Number	Date	Editor/ Writer	Substantive Change(s)
0	11/2006	DM	Initial release.

Document Number: AN3099
Rev. 0
11/2006