```
 if (CheckFrameReceived())                  // Packet received
 {
        if (IsBroadcast()) {
                        ProcessEthBroadcastFrame();
                          } else {
                                ProcessEthIAFrame();
                                 }
  EndReadFrame();                           // release buffer in ethernet controller
 }
```

```
unsigned int CheckFrameReceived(void) {          // Packet received ?
 // more packets received ?
                if (LPC_EMAC->RxProduceIndex != LPC_EMAC->RxConsumeIndex)
                return(1);
                        else
                return(0);
}
```

```
unsigned int IsBroadcast(void) {
  unsigned short RecdDestMAC[3];        // 48 bit MAC
          RecdFrameLength = StartReadFrame();
// receive Destination Address to see if it was a broadcast
          CopyFromFrame_EMAC(&RecdDestMAC,  6);
// store Source Address (for our answer)
          CopyFromFrame_EMAC(&RecdFrameMAC, 6);

                if ((RecdDestMAC[0] == 0xFFFF) &&
                    (RecdDestMAC[1] == 0xFFFF) &&
                    (RecdDestMAC[2] == 0xFFFF)) {
                                        return(1);
                                                } else {
                                        return (0);
                                                }
}
```

```
void CopyFromFrame_EMAC(void *Dest, unsigned short Size)
{       unsigned short * piDest;            // Keil: Pointer added to correct expression
        piDest = Dest;                      // Keil: Line added
  while (Size > 1) {
                *piDest++ = ReadFrame_EMAC();
                 Size -= 2;
```

```
                }

  if (Size) {                           // check for leftover byte...
                                        // the LAN-Controller will return 0
                                        // for the highbyte
        *(unsigned char *)piDest = (char)ReadFrame_EMAC();
          }
}
```

```
// reads a word in little-endian byte order from RX_BUFFER
unsigned short ReadFrame_EMAC(void)
{
  return (*rptr++);
}
```

```
// easyWEB internal function
// handles an incoming broadcast frame
void ProcessEthBroadcastFrame(void)
{       unsigned short TargetIP[2];

  if (ReadFrameBE_EMAC() == FRAME_ARP)          // get frame type, check for ARP
    if (ReadFrameBE_EMAC() == HARDW_ETH10)      // Ethernet frame
      if (ReadFrameBE_EMAC() == FRAME_IP)       // check protocol
        if (ReadFrameBE_EMAC() == IP_HLEN_PLEN) // check HLEN, PLEN
          if (ReadFrameBE_EMAC() == OP_ARP_REQUEST)
          {
                DummyReadFrame_EMAC(6);             // ignore sender's hardware address
                CopyFromFrame_EMAC(&RecdFrameIP, 4); // read sender's protocol address
                DummyReadFrame_EMAC(6);             // ignore target's hardware address
                CopyFromFrame_EMAC(&TargetIP, 4);   // read target's protocol address
                 if (!memcmp(&MyIP, &TargetIP, 4))  // is it for us?
                PrepareARP_ANSWER();                // yes->create ARP_ANSWER frame
          }
}
```

```
// reads a word in big-endian byte order from RX_FRAME_PORT
// (useful to avoid permanent byte-swapping while reading
// TCP/IP-data)
unsigned short ReadFrameBE_EMAC(void)
{       unsigned short ReturnValue;
```

```
        ReturnValue = SwapBytes (*rptr++);
        return (ReturnValue);
}
```

```
// easyWEB internal function
// help function to swap the byte order of a uint16_t
unsigned short SwapBytes(unsigned short Data)
{
  return (Data >> 8) | (Data << 8);
}
```

```
// does a dummy read on frame-I/O-port
// NOTE: only an even number of bytes is read!
void DummyReadFrame_EMAC(unsigned short Size)    // discards an EVEN number of bytes
{                                                // from RX-fifo
        while (Size > 1) {
                        ReadFrame_EMAC();
                        Size -= 2;
                          }
}
```

```
extern _ARMABI int memcmp(const void * /*s1*/, const void * /*s2*/, size_t /*n*/)
__attribute__((__nonnull__(1,2)));
  /*
   * compares the first n characters of the object pointed to by s1 to the
   * first n characters of the object pointed to by s2.
   * Returns: an integer greater than, equal to, or less than zero, according
   *         as the object pointed to by s1 is greater than, equal to, or
   *         less than the object pointed to by s2.
   */
```

```
void PrepareARP_ANSWER(void)
{
  // Ethernet
  memcpy(&TxFrame2[ETH_DA_OFS], &RecdFrameMAC, 6);
  memcpy(&TxFrame2[ETH_SA_OFS], &MyMAC, 6);
  *(unsigned short *)&TxFrame2[ETH_TYPE_OFS] = SWAPB(FRAME_ARP);

  // ARP
  *(unsigned short *)&TxFrame2[ARP_HARDW_OFS] = SWAPB(HARDW_ETH10);
```

```
  *(unsigned short *)&TxFrame2[ARP_PROT_OFS] = SWAPB(FRAME_IP);
  *(unsigned short *)&TxFrame2[ARP_HLEN_PLEN_OFS] = SWAPB(IP_HLEN_PLEN);
  *(unsigned short *)&TxFrame2[ARP_OPCODE_OFS] = SWAPB(OP_ARP_ANSWER);
  memcpy(&TxFrame2[ARP_SENDER_HA_OFS], &MyMAC, 6);
  memcpy(&TxFrame2[ARP_SENDER_IP_OFS], &MyIP, 4);
  memcpy(&TxFrame2[ARP_TARGET_HA_OFS], &RecdFrameMAC, 6);
  memcpy(&TxFrame2[ARP_TARGET_IP_OFS], &RecdFrameIP, 4);

  TxFrame2Size = ETH_HEADER_SIZE + ARP_FRAME_SIZE;
  TransmitControl |= SEND_FRAME2;
}
```

```
// easyWEB internal function
// handles an incoming frame that passed EMAC's address filter
// (individual addressed = IA)
void ProcessEthIAFrame(void)
{       unsigned short TargetIP[2];
        unsigned char ProtocolType;
  switch (ReadFrameBE_EMAC())                 // get frame type
  {
   case FRAME_ARP :                         // check for ARP
   {
    if ((TCPFlags & (TCP_ACTIVE_OPEN | IP_ADDR_RESOLVED)) ==
TCP_ACTIVE_OPEN)
      if (ReadFrameBE_EMAC() == HARDW_ETH10)        // check for the right prot. etc.
       if (ReadFrameBE_EMAC() == FRAME_IP)
        if (ReadFrameBE_EMAC() == IP_HLEN_PLEN)
         if (ReadFrameBE_EMAC() == OP_ARP_ANSWER)
         {
          TCPStopTimer();                   // OK, now we've the MAC we wanted ;-)
          CopyFromFrame_EMAC(&RemoteMAC, 6);    // extract opponents MAC
          TCPFlags |= IP_ADDR_RESOLVED;
         }
    break;
   }
   case FRAME_IP :                              // check for IP-type
   {                                           // IPv4, IHL=5 (20 Bytes Header)
    if ((ReadFrameBE_EMAC() & 0xFF00 ) == IP_VER_IHL)
    {                                       // ignore Type Of Service
     RecdIPFrameLength = ReadFrameBE_EMAC();          // get IP frame's length
     ReadFrameBE_EMAC();                        // ignore identification
       // only unfragm. frames
     if (!(ReadFrameBE_EMAC() & (IP_FLAG_MOREFRAG | IP_FRAGOFS_MASK)))
```

```
      {
       ProtocolType = ReadFrameBE_EMAC() & 0xFF;        // get protocol, ignore TTL
       ReadFrameBE_EMAC();                              // ignore checksum
       CopyFromFrame_EMAC(&RecdFrameIP, 4);             // get source IP
       CopyFromFrame_EMAC(&TargetIP, 4);                // get destination IP

       if (!memcmp(&MyIP, &TargetIP, 4))                // is it for us?
        switch (ProtocolType) {
          case PROT_ICMP : { ProcessICMPFrame(); break; }
          case PROT_TCP  : { ProcessTCPFrame(); break; }
          case PROT_UDP  : break;                       // not implemented!
        }
      }
     }
     break;
   }
  }
}
```

```
// easyWEB internal function
// we've just rec'd an ICMP-frame (Internet Control Message Protocol)
// check what to do and branch to the appropriate sub-function
void ProcessICMPFrame(void)
{       unsigned short ICMPTypeAndCode;
 ICMPTypeAndCode = ReadFrameBE_EMAC();          // get Message Type and Code
 ReadFrameBE_EMAC();                            // ignore ICMP checksum
 switch (ICMPTypeAndCode >> 8) {                // check type
  case ICMP_ECHO :                              // is echo request?
  {
   PrepareICMP_ECHO_REPLY();                    // echo as much as we can...
   break;
  }
 }
}
```

```
void EndReadFrame(void) {
 unsigned int idx;
        /* DMA free packet. */
                idx = LPC_EMAC->RxConsumeIndex;
                if (++idx == NUM_RX_FRAG) idx = 0;
                 LPC_EMAC->RxConsumeIndex = idx;
}
```

```
void TCPLowLevelInit(void)
{
  // Keil: Timer 0 is used for TCP retransmission control
  LPC_TIM0->MR0 = 3144000;     // 262mSec
  LPC_TIM0->MCR = 3;           // Interrupt and Reset on MR0
  LPC_TIM0->TCR = 1;           // Timer0 Enable
  NVIC_EnableIRQ(TIMER0_IRQn);

  LPC_SC->PCONP |= (1<<12);    // Deliver clock to AD
  /* all the related pins are set to ADC inputs, AD0.0~7 */
  LPC_PINCON->PINSEL0 |= 0x0F000000;       /* P0.12~13, A0.6~7, function 11 */
  LPC_PINCON->PINSEL1 &= ~0x003FC000;  /* P0.23~26, A0.0~3, function 01 */
  LPC_PINCON->PINSEL1 |= 0x00154000;
  LPC_PINCON->PINSEL3 |= 0xF0000000;       /* P1.30~31, A0.4~5, function 11 */

  Init_EMAC();
  TransmitControl = 0;
  TCPFlags = 0;
  TCPStateMachine = CLOSED;
  SocketStatus = 0;
}
```

```
// configure port-pins for use with LAN-controller,
// reset it and send the configuration-sequence
void Init_EMAC(void)
{
// Keil: function modified to access the EMAC
// Initializes the EMAC ethernet controller
  unsigned int regv,tout,id1,id2;
  /* Power Up the EMAC controller. */
  LPC_SC->PCONP |= (0x1<<30);

  LPC_PINCON->PINSEL2 = 0x50150105;
#if MDC_MDIO_WORKAROUND
  /* LPC175x devices, use software MII management. */
  LPC_PINCON->PINSEL4 &= ~0x000F0000;
  LPC_GPIO2->FIODIR |= MDC;
#else
  LPC_PINCON->PINSEL3 &= ~0x0000000F;
  LPC_PINCON->PINSEL3 |= 0x00000005;
#endif
  /* Reset all EMAC internal modules. */
```

```
  LPC_EMAC->MAC1 = MAC1_RES_TX | MAC1_RES_MCS_TX | MAC1_RES_RX |
MAC1_RES_MCS_RX | MAC1_SIM_RES | MAC1_SOFT_RES;
  LPC_EMAC->Command = CR_REG_RES | CR_TX_RES | CR_RX_RES;
  /* A short delay after reset. */
  for (tout = 100; tout; tout--);
  /* Initialize MAC control registers. */
  LPC_EMAC->MAC1 = MAC1_PASS_ALL;
  LPC_EMAC->MAC2 = MAC2_CRC_EN | MAC2_PAD_EN;
  LPC_EMAC->MAXF = ETH_MAX_FLEN;
  LPC_EMAC->CLRT = CLRT_DEF;
  LPC_EMAC->IPGR = IPGR_DEF;
  /* Enable Reduced MII interface. */
  LPC_EMAC->Command = CR_RMII | CR_PASS_RUNT_FRM;
  /* Reset Reduced MII Logic. */
  LPC_EMAC->SUPP = SUPP_RES_RMII;
  for (tout = 100; tout; tout--);
  LPC_EMAC->SUPP = 0;
  /* Put the DP83848C in reset mode */
  write_PHY (PHY_REG_BMCR, 0x8000);
  /* Wait for hardware reset to end. */
  for (tout = 0; tout < 0x100000; tout++) {
    regv = read_PHY (PHY_REG_BMCR);
    if (!(regv & 0x8000)) {
      /* Reset complete */
      break;
    }
  }
  /* Check if this is a DP83848C PHY. */
  id1 = read_PHY (PHY_REG_IDR1);
  id2 = read_PHY (PHY_REG_IDR2);
  if (((id1 << 16) | (id2 & 0xFFF0)) == DP83848C_ID) {
          /* Configure the PHY device */
           /* Use autonegotiation about the link speed. */
           write_PHY (PHY_REG_BMCR, PHY_AUTO_NEG);
           /* Wait to complete Auto_Negotiation. */
           for (tout = 0; tout < 0x100000; tout++) {
           regv = read_PHY (PHY_REG_BMSR);
          if (regv & 0x0020) {
           /* Autonegotiation Complete. */
          break;
          }
    }
  }
  /* Check the link status. */
```

```
  for (tout = 0; tout < 0x10000; tout++) {
    regv = read_PHY (PHY_REG_STS);
    if (regv & 0x0001) {
      /* Link is on. */
      break;
    }
  }
  /* Configure Full/Half Duplex mode. */
  if (regv & 0x0004) {
    /* Full duplex is enabled. */
    LPC_EMAC->MAC2    |= MAC2_FULL_DUP;
    LPC_EMAC->Command |= CR_FULL_DUP;
    LPC_EMAC->IPGT     = IPGT_FULL_DUP;
  }
  else {
    /* Half duplex mode. */
    LPC_EMAC->IPGT = IPGT_HALF_DUP;
  }
  /* Configure 100MBit/10MBit mode. */
  if (regv & 0x0002) {
    /* 10MBit mode. */
    LPC_EMAC->SUPP = 0;
  }
  else {
    /* 100MBit mode. */
    LPC_EMAC->SUPP = SUPP_SPEED;
  }
  /* Set the Ethernet MAC Address registers */
  LPC_EMAC->SA0 = (MYMAC_6 << 8) | MYMAC_5;
  LPC_EMAC->SA1 = (MYMAC_4 << 8) | MYMAC_3;
  LPC_EMAC->SA2 = (MYMAC_2 << 8) | MYMAC_1;
  /* Initialize Tx and Rx DMA Descriptors */
  rx_descr_init ();
  tx_descr_init ();
  /* Receive Broadcast and Perfect Match Packets */
  LPC_EMAC->RxFilterCtrl = RFC_BCAST_EN | RFC_PERFECT_EN;
  /* Enable EMAC interrupts. */
  LPC_EMAC->IntEnable = INT_RX_DONE | INT_TX_DONE;
  /* Reset all interrupts */
  LPC_EMAC->IntClear  = 0xFFFF;
  /* Enable receive and transmit mode of MAC Ethernet core */
  LPC_EMAC->Command  |= (CR_RX_EN | CR_TX_EN);
  LPC_EMAC->MAC1     |= MAC1_REC_EN;
}
```

```
// Keil: function added to initialize Rx Descriptors
void rx_descr_init (void)
{ unsigned int i;
        for (i = 0; i < NUM_RX_FRAG; i++) {
         RX_DESC_PACKET(i)  = RX_BUF(i);
        RX_DESC_CTRL(i)    = RCTRL_INT | (ETH_FRAG_SIZE-1);
        RX_STAT_INFO(i)    = 0;
        RX_STAT_HASHCRC(i) = 0;
        }
 /* Set EMAC Receive Descriptor Registers. */
 LPC_EMAC->RxDescriptor    = RX_DESC_BASE;
 LPC_EMAC->RxStatus        = RX_STAT_BASE;
 LPC_EMAC->RxDescriptorNumber = NUM_RX_FRAG-1;
 /* Rx Descriptors Point to 0 */
 LPC_EMAC->RxConsumeIndex  = 0;
}
```