AN10775

NicheLite for LPC implementation notes

Rev. 01 — 19 December 2008 Application note

Document information

Info	Content
Keywords	Network, Ethernet, TCP/IP Stack, LPC2400, LPC3250
Abstract	This application note discusses implementation details when using the NicheLite for LPC TCP/IP stack in a project. Aspects discussed include memory management, stack operation and customization.

founded by Philips

Revision history

Rev	Date	Description
01	20081219	First version.

Contact information

For additional information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

1. Introduction

Several members of the NXP LPC2000 series flash microcontrollers provide an on-chip Ethernet controller, allowing easy connection to a network. NXP makes available for LPC2000 users the NicheLite for LPC TCP/IP stack. This stack is a cut down version of the full stack available from InterNiche.

The advantage of using the NicheLite for LPC stack is that it is royalty free and it is already ported to the LPC2000 family. However embedded engineers still have to integrate the stack with their application. Integration considerations include performance and memory usage.

This application note assumes basic knowledge of TCP/IP protocols, such as IP, TCP and UDP. It is intended to complement the documentation from InterNiche, and not replace any such documentation.

2. Packets

This section describes aspects related to packets. Data is transferred across the network in packets, which can be up to the Maximum Transfer Unit (MTU) in size.

2.1 Packet Allocation and Deallocation

All data in and out of the TCP/IP stack is transferred using packets. Packets contain the Ethernet header, the IP header and additional data depending on the protocols used. The packets are allocated from the heap when the stack starts executing and remain allocated for the runtime of the firmware. All packets are placed into a pool of free packets after allocation. They are moved out of the free pool when they are used by the stack and back into the free pool when they are no longer needed by the stack.

2.1.1 Packet Types and Configuration

There are two types of packet in the NicheLite implementation, big and little (“lil” in the stack source code). The number and maximum size of each type is defined in ipport.h.

Table 1. Packet Configuration Identifiers

Identifier	Description
NUMBIGBUFS	Total number of big packets
BIGBUFSIZE	Size of big packets in octets
NUMLILBUFS	Total number of little packets
LILBUFSIZE	Size of little packets in octets

When a packet is needed by a protocol pk_alloc is called. This function uses a little packet if the data size for the packet is small enough, otherwise it uses a big packet. If all the little packets are in use then a big packet will be used.

The identifiers can be adjusted to optimize packet usage for the specific embedded system. If a lot of small packets will be sent then it may be beneficial to increase the number of little packets and reduce the number of big packets, for example.

2.2 Increasing the Total Number of Packets

The maximum packet size (MTU) is set to a typical value, such as 1514 bytes. This allows data up to the MTU in size be transmitted across most networks without fragmentation.

It can be desirable to increase the number of packets available in an embedded system to allow for increased concurrent communications to take place. Care must be taken to ensure that at no point in time all packets are in use and another is needed.

For example an embedded system may have a TCP based control channel, an IGMP management system and a TELNET based interface. In the worst case all three communication interfaces will be transmitting and receiving packets at the same time. The problem becomes more critical if the embedded system needs to stream unconfirmed packets of data.

The solution is to increase the number of packets available to the stack. This is possible by decreasing the maximum packet size that the embedded system can handle.

By changing the MTU the system will not be able to transmit or receive packets above that size, but more packets can fit into the available memory. Configuring the maximum packet size in the NicheLite stack configures the Ethernet Maximum Frame Register (MAXF), which will cause the Ethernet controller to reject packets above this size without interrupting the CPU.

How can the maximum packet size be determined? This is achieved by examining the existing protocols that will be needed, such as DHCP, along with careful design of custom protocols. The following table indicates the typical data sizes for commonly used protocols.

Table 2. Typical Protocol Data Sizes

Protocol	Data Size (octets)
DHCP (must have a minimum of 312 octets of options)	548
ARP	28

To these values the UDP header, IP header and Ethernet header lengths must be added to get the largest packet size. Some examples are in the following table.

Table 3. Typical Header Sizes

Protocol	Header Size (octets)
UDP	8
IP	24 (depending on options used)
Ethernet	22

For example DHCP packets might be 548 + 8 + 24 + 22 = 602 octets in size. Note that not all protocols are built upon UDP.

2.2.1 Stack Configuration

To change the maximum size of packets supported by the Ethernet controller and the stack:

- In ipport.h change BIGBUFSIZE to the maximum packet size, including all headers

- In ether.h change MTU and ET_MAXLEN to BIGBUFSIZE minus 22 (the size of the Ethernet header

2.2.2 Testing

It is recommended to stress test the embedded system in order to produce the worst case scenario in terms of packets in use at any moment in time.

2.3 Runtime Monitoring

The usage of packets can be monitored during runtime when using a suitable debug interface, such as JTAG or an In-Circuit Emulator.

Access to some internal data is also available via the default serial interface system, however because the interface may affect performance and may not be needed or desired for production firmware, it may be disabled. Using a system such as JTAG provides more flexibility during debugging, and is recommended.

2.3.1 Detailed Packet Usage

In ipport.h ensure NDEBUG is defined. This allows access to the pktlog array, defined in pktalloc.c. Create a watchpoint on pktlog to access the contents when execution is stopped.

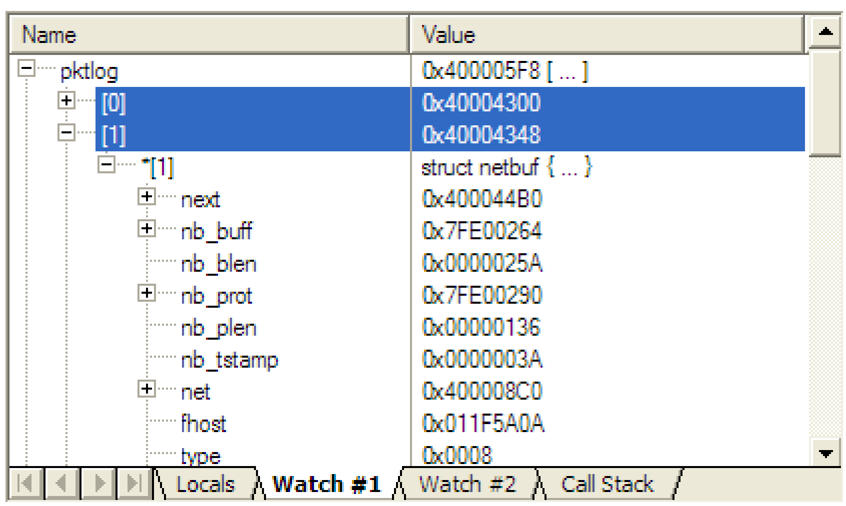

Fig 1. Watching pktlog

The first NUMBIGBUFS (see ipport.h) entries in the array are the big packets. The remaining entries are the little packets. For example if NUMBIGBUFS is set to 10, then packets zero to nine will be big, followed by the little packets.

The packet data is located at nb_buff. This includes Ethernet and IP headers. Higher layer protocol data is stored at m_data, and has a length of m_len. If a packet is currently in use by the stack then the inuse member will be non-zero. The fhost member contains the IP address of the remote host.

By looking at fhost, inuse, m_data and m_len it is possible to understand the purpose of all the packets in the system when execution is paused.

All members of the netbuf structure are briefly described in netbuf.h.

2.3.2 Total Packet Count

By adding a few lines of code to pktalloc.c it is possible to make debugging packet usage a little easier. The method involves creating two variables to hold the current number of packets of each type in use.

In pktalloc.c create two variables to keep count.

```
int bigbuffree = NUMBIGBUFS;
int lilbuffree = NUMLILBUFS;
```

In the pkt_alloc function add two lines to decrement the counters.

```
if ((len > lilbufsiz) || (lilfreeq.q_len == 0)) /* must use a big buffer */
{
  p = (PACKET)getq(&bigfreeq);
  if (p) bigbuffree--;
}
else
{
  p = (PACKET)getq(&lilfreeq);
  if (p) lilbuffree--;
}
```

In the pkt_free function add two lines to increment the counters.

```
if (pkt->nb_blen == bigbufsiz)
{
  q_add(&bigfreeq, (qp)pkt);
  bigbuffree++;
}
else
{
  q_add(&lilfreeq, (qp)pkt);
  lilbuffree++;
}
```

The counters can be watched during runtime to monitor the usage of both packet types under various conditions.

2.3.3 Packet Statistics

The NicheLite stack provides a structure containing statistics on the packets. This includes such aspects as the number of packets transmitted and received and the number of packets discarded. This is very useful for determining if there might be unseen data processing problems.

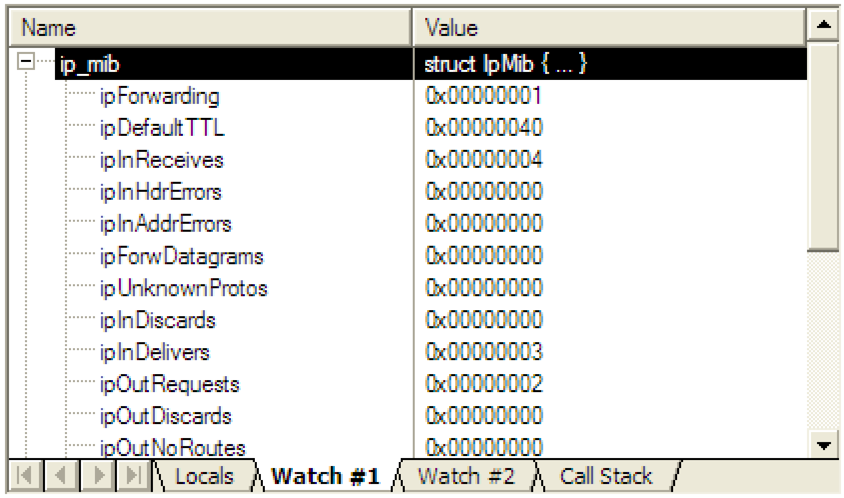

Fig 2. Watching ip_mib

The ip_mib structure is defined in m_ip.c.

3. TCP

This section describes various aspects of the TCP implementation in the NicheLite stack and how to accomplish specific tasks.

3.1 Retransmits

If the embedded system periodically sends segments of data to a remote host, and the segments are not acknowledged, then they are automatically retransmitted up to 12 times. The time between retransmits increases each time. During this time the memory remains allocated for the Ethernet packet containing the segment.

TCP has the ability to transmit additional segments while handling retransmits for other segments. If the connection is broken or the remote host crashes, there could be an increasing backlog of segments going through the retransmit process. In the worst case all Ethernet packet buffers will be exhausted. This problem could be exacerbated if the embedded system periodically transmits TCP segments.

It is possible to change the number of retransmits using the TCP_MAXRXTSHIFT identifier defined in mtcp.h. Whether this is a suitable change depends on the application being developed.

It is recommended that a TCP based protocol implement a command/response sequence for controlling the flow of data. This would allow failure of the network or remote host to be detected before additional TCP segments are transmitted.

Note that closing a connection causes the stack to free all pending packets for the connection.

An alternative method is to use TCP keepalives to detect a broken connection.

3.2 Keepalives

Keepalives are small messages periodically sent between two hosts to inform each end that the connection is operational.

Suppose a remote host opens a TCP connection to the embedded system. While the connection is open there is no indication that the connection is functional unless data is sent. If the remote host crashes or the network connection is broken the embedded system will not know about it. This can cause two potential problems.

- Any packets in the process of being assembled for transmission to the remote host can get stuck in limbo, causing a memory leak
- The socket for the connection to the remote host is never closed, causing a memory leak

Although the memory leak may be slow, if the embedded system is required to have 100% uptime, the system could run out of RAM after months of operation.

There are three aspects to using TCP keepalives in the NicheLite stack.

- How to enable the feature for specific sockets
- The default timing follows RFC1122, which is probably too long for most embedded systems
- There is a minor issue in the implementation in some versions of the stack

The following subsections deal with each of these aspects in turn.

3.2.1 Enabling Keepalives

Typically socket options are enabled and disabled using the m_ioctl function, however the NicheLite implementation of this function does not support enabling keepalives. Therefore the feature must be enabled directly. Alternatively the m_ioctl function is easily modified to add this option.

Calling m_listen or m_connect returns a pointer to a socket. The option is enabled by using the so_options member.

```
WP_SOCKTYPE sock;
sock = m_listen(&sin, upcall, &e);
// enable keep alives
sock->so_options |= SO_KEEPALIVE;
```

3.2.2 Adjusting Timing

As previously mentioned, the default timing for keepalives follows the Internet standard RFC1122. The timing values are defined in mtcp.h and are listed in the following table.

Table 4. TCP Keepalive Timing Identifiers

Identifier	Description
TCPTV_KEEP_INIT	The time during which the connection must be established
TCPTV_KEEP_IDLE	Default time before probing
TCPTV_KEEPINTVL	Default probe interval

Identifier	Description
TCPTV_KEEPCNT	Max probes before drop

When modifying these values be aware that TCPTV_KEEP_INIT may be defined twice, depending on the version of the NicheLite stack.

As an example, the default value of TCPTV_KEEP_IDLE is two hours. This means that if no data is received for two hours then keepalives will be transmitted. In a typical embedded system a more appropriate value might be one minute.

The number of keepalive probes that go unacknowledged before the stack drops the connection is also configurable. The best strategy is probably to work out the maximum time that the system can commit RAM to a dead connection, and then work out appropriate keepalive values to achieve that requirement.

Note that these timing settings apply to all connections that have keepalives enabled.

3.2.3 Fixing Sequence Number Issue

This section might not apply to your version of the NicheLite stack.

RFC1122 states that the keepalive message must have a sequence number of one less than the next expected sequence number. The NicheLite stack may send the next sequence number instead. Those keepalives will not be acknowledged by some TCP/IP stacks, including the implementation in Windows XP.

In function tcp_output in tcpout.c, change the following lines

```
/* set seq for a BSD-ish keepalive */
ptcp->th_seq = ((tp->snd_nxt >> 24) & 0x00ff) |
               ((tp->snd_nxt >>  8) & 0xff00) |
               ((tp->snd_nxt & 0xff00) <<  8) |
               ((tp->snd_nxt & 0x00ff) << 24);
```

to the following

```
/* set seq for a BSD-ish keepalive */
ptcp->th_seq = (((tp->snd_nxt - 1) >> 24) & 0x00ff) |
               (((tp->snd_nxt - 1) >>  8) & 0xff00) |
               (((tp->snd_nxt - 1) & 0xff00) <<  8) |
               (((tp->snd_nxt - 1) & 0x00ff) << 24);
```

3.2.4 Detecting Dropped Connections

When a connection is dropped because keepalives were not being acknowledged the TCP callback function will be called with an event code of M_CLOSED. The socket error will be set to ETIMEDOUT.

```
case M_CLOSED:
  if (so->error == ETIMEDOUT)
  {
```

```
      // connection dropped by stack
    }
    break;
```

Note that there is no need to call m_close as the socket will automatically be closed by the stack.

4. UDP

This section contains brief notes on using the UDP layer.

4.1 Return Values When Sending UDP Datagrams

The first time udp_send is called for a specific host the stack may have to go through the ARP process to find out information about the host. If this is the case then the udp_send function will return ENP_SEND_PENDING. This is defined with the value of one, and isn’t an error condition. Therefore to check for errors returned by the function only negative numbers should be considered.

```
If (udp_send(port1, port2, packet) < 0)
{
  // UDP send error
}
```

4.2 UDP Callback Function

To avoid a memory leak, the udp_free function must be called in the UDP callback function when the packet of data has been processed. This is demonstrated in the DHCP client example supplied with the NicheLite stack.

5. Sockets

This section contains notes and issues relating to the use of sockets. When a connection is opened or listened for, a socket is created. The socket is a data structure describing the status of the connection and providing a means of transmitting to a remote host.

5.1 Freeing Allocated Memory

When a connection is closed the memory allocated to the description of the socket must be freed. It is possible that there is a delay between the connection closing and the memory being freed. If the embedded system is handling enough connection and disconnection requests there could be a memory leak as the list of pending sockets to be freed increases.

It is possible to force the stack to free the memory for the socket immediately. This is handled in the TCP callback function when the event code is M_CLOSED. The following code demonstrates how this is achieved.

```
case M_CLOSED:
  if (so->error != ETIMEDOUT)
```

```
{
  so->so_options |= SO_LINGER;
  so->linger = 0;
}
break;
```

Note that the SO_LINGER option should not be used if the stack closed the connection due to a timeout condition.

5.2 Runtime Monitoring

The usage of sockets can be monitored during runtime when using a suitable debug interface, such as JTAG or an In-Circuit Emulator.

Access to some internal data is also available via the default serial interface system, however because the interface may affect performance and may not be needed or desired for production firmware, it may be disabled. Using a system such as JTAG provides more flexibility during debugging, and is recommended.

5.2.1 Socket Usage

Details of how many sockets are in use and what for may be obtained by watching the msoq linked list defined in tcputil.c.

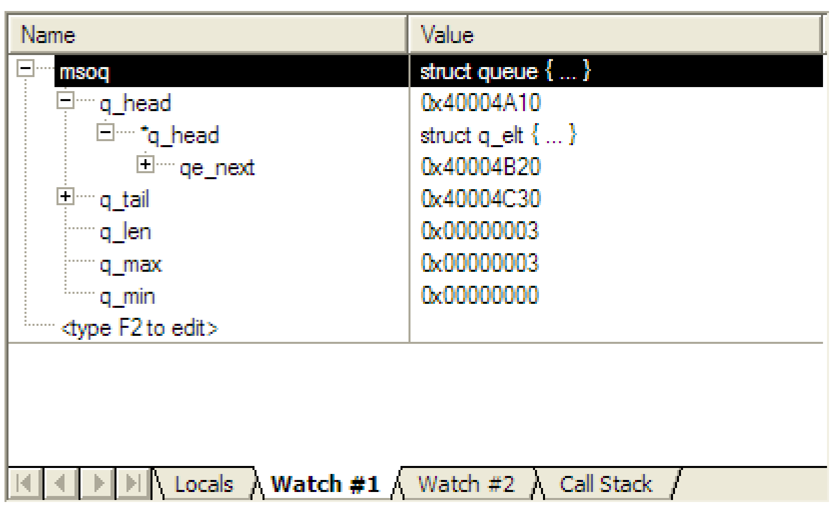

Fig 3. Watching msoq

The member q_len is the size of the list, i.e. the number of sockets in use. The q_max member is the highest number of sockets that have been in use since the stack started executing.

6. Stack Configuration

It may be necessary to configure aspects of how the TCP/IP stack behaves for tight integration into an application. This section describes some aspects of this process.

6.1 IP Address Configuration

A common requirement for embedded systems that use a TCP/IP stack is to allow for the end user to configure the IP address, and whether the IP address is fixed (static) or obtained from the network (dynamic) using a DHCP server.

For example the system may provide a HTTP or TELNET based interface allowing the IP address to be configured. This is necessary due to the wide variation in networks that the system may be installed into.

The configuration of the IP address is performed in the function pre_task_setup in in_stubs.c. The following is a code snippet of the relevant section.

```
#ifdef DHCP_CLIENT
netstatic[i].n_flags |= NF_DHCPC;
netstatic[i].n_ipaddr = 0x00000000;    /* 0.0.0.0 */
#else
netstatic[i].n_ipaddr = 0x6400000A;    /* 10.0.0.100 */
#endif
netstatic[i].snmask   = 0x000000FF;    /* 255.0.0.0 */
netstatic[i].n_defgw  = 0x0100000A;    /* 10.0.0.1 */
i++;
```

This section of code configures the netstatic array for the Ethernet interface number i. If a dynamic IP address is to be used then NF_DHCPC is ORed with n_flags and the n_ipaddr value is ignored. If a static IP address is to be used then it is placed into n_ipaddr.

The snmask contains the IP network mask and n_defgw contains the default gateway for the network.

To allow these values to be configurable these lines need to be replaced with code that can obtain the current configuration, perhaps from non-volatile memory. The following is an example for illustrative purposes.

```
// configure static/dynamic ip address based on nvol settings
if (NVol_UseStaticIP())
{
  NVol_GetStaticIP(staticip);
  netstatic[i].n_ipaddr = ((unsigned long)staticip[3] << 24) |
                          ((unsigned long)staticip[2] << 16) |
                          ((unsigned long)staticip[1] << 8)  |
                           (unsigned long)staticip[0];
}
else
{
  netstatic[i].n_flags |= NF_DHCPC;
  netstatic[i].n_ipaddr = 0x00000000;
}
i++;
```

6.2 MAC Address Configuration

Every Ethernet controller must have a unique MAC address, which is assigned by the IEEE, and this value must be supplied to the TCP/IP stack.

The default hard-coded MAC address for the NicheLite stack is defined in the eth_info array, which is initialized in emac.c. Because it typically isn't practical to compile different source code for each copy of a product, this method will likely need to be adjusted. For example the eth_info array could be located at a specific location in flash memory, allowing patching of the hex file on the production line.

An alternative approach is to store the MAC address in non-volatile memory and supply it to the stack during initialization. This would allow for more flexibility in configuration of the MAC address.

The MAC address should be assigned to the mac_addr member of the eth_info array inside the eth_init function in emac.c. The following code snippet is an example for illustrative purposes.

```
// get MAC address from NVol module
if (!NVol_GetMACAddress((unsigned char *)&(eth_info[dev].mac_addr)))
{
  dprintf("*** failed to get mac address, cannot init ethernet\n");
  return (-1);
}
```

Note that this should be performed before the mac_addr member is used for the first time in the eth_init function.

7. Implementing Raw Send Functionality

Embedded systems often need access to lower level functionality, especially in situations were optimization is required or out-of-the-ordinary operations need to take place.

The NicheLite stack does not provide a single interface for the application to construct IP datagrams from scratch, but it is possible. Note that this approach relies on the internal data structures used by the stack, which could change in future versions.

7.1 Example For A Custom Protocol

The following code listing demonstrates how to send an IP datagram containing a custom protocol by directly constructing an IP datagram. The protocol is called MYPROT and contains some data along with an IP-style checksum.

```
PACKET p;
struct arptabent tp;
struct ip *ip_header;
char *pprot;
struct myprot *myprot_data;

// allocate memory for packet
LOCK_NET_RESOURCE(FREEQ_RESID);
p = pk_alloc(ETHHDR_SIZE + sizeof(struct ip) + IP_OPTIONS_LEN + sizeof(struct myprot));
```

```
UNLOCK_NET_RESOURCE(FREEQ_RESID);
if (!p)
{
  dprintf("Failed to allocate memory for MYPROT packet\n");
  return;
}

// get start of IP/protocol data buffer
p->nb_prot = p->nb_buff + ETHHDR_SIZE;
// set length of IP/MYPROT data
p->nb_plen = sizeof(struct ip) + IP_OPTIONS_LEN + sizeof(struct myprot);
// set destination host
p->fhost = MYPROT_HOST;
// use first ethernet interface
p->net = nets[0];

// store IP header
ip_header = (struct ip *)p->nb_prot;
ip_header->ip_ver_ihl   = 4 << 4;                                   // IP version 4
ip_header->ip_ver_ihl  |= (sizeof(struct ip) + IP_OPTIONS_LEN) / 4;  // length of IP header in 32-bit words
ip_header->ip_tos       = 0x00;                                      // default
ip_header->ip_len       = HTONS(24 + sizeof(struct myprot));         // total message length
ip_header->ip_id        = (unshort)((uid >> 8) | (uid << 8));        // IP datagram ID
ip_header->ip_flgs_foff = 0x0000;                                    // no flags
ip_header->ip_time      = 64;                                        // TTL
ip_header->ip_prot      = MYPROT_PROT;                               // MYPROT message
ip_header->ip_chksum    = IPXSUM;                                    // checksum - zero for calculation
ip_header->ip_src       = m_netp->n_ipaddr;                          // our IP address
ip_header->ip_dest      = MYPROT_HOST;                               // destination
// add IP options (IP_OPTIONS_LEN bytes in size)
pprot = p->nb_prot + sizeof(struct ip);
pprot[0] = 0x94;                                                     // router alert option
pprot[1] = 0x04;
pprot[2] = 0x00;
pprot[3] = 0x00;
pprot += IP_OPTIONS_LEN;

// calculate and store IP header checksum
ip_header->ip_chksum = ~cksum(ip_header, sizeof(struct ip) + IP_OPTIONS_LEN);

// increment id for next datagram
uid++;

// store MYPROT data
myprot_data = (struct myprot *)pprot;
myprot_data->type          = 0x01;
myprot_data->resptime      = 0x02;
myprot_data->checksum      = IPXSUM;

// calculate and store MYPROT checksum
```

```
60   myprot_data->checksum = ~cksum(myprot_data, sizeof(struct myprot));
61
62   // destination MAC address
63   tp.t_phy_addr[0] = 0x01;
64   tp.t_phy_addr[1] = 0x00;
65   tp.t_phy_addr[2] = 0x5E;
66   tp.t_phy_addr[3] = 0x7F;
67   tp.t_phy_addr[4] = 0xFF;
68   tp.t_phy_addr[5] = 0xFF;
69
70   // add ethernet header and send it
71   if (et_send(p, &tp) != 0)
72   {
73     dprintf("Failed to send MYPROT message\n");
74     return;
75   }
```

Line 9: allocates memory for the packet. The total size is made up of the Ethernet header, IP header, IP options and MYPROT data.

Lines 18 - 24: some members of the packet need to be initialized. nb_prot is set to point to the start of the IP header.

Lines 27 – 44: configure the IP header for the message. This includes a TTL value and the protocol number for the custom protocol.

Lines 54 – 57: the data is filled in for the custom protocol.

Lines 63 – 68: the MAC address for the destination is filled in. Alternatively the stack can be instructed to go through the ARP process and automatically determine the destination MAC address.

8. Versions Used

This application note was developed with the following software versions:

- NicheLite for LPC revision 1.02
- Keil RealView MDK-ARM version 3.20

9. Legal information

9.1 Definitions

Draft — The document is a draft version only. The content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included herein and shall have no liability for the consequences of use of such information.

9.2 Disclaimers

General — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in medical, military, aircraft, space or life support equipment, nor in applications where failure or malfunction of a NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors accepts no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is for the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

9.3 Trademarks

Notice: All referenced brands, product names, service names and trademarks are property of their respective owners.

10. Contents

For more information, please visit: http://www.nxp.com
For sales office addresses, email to: salesaddresses@nxp.com

Date of release: 19 December 2008

Document identifier: AN10775_1