UCLA WAMISNOS
Network Protocol Programmers Guide*

Release 1.3
(March 1995)

Walter A. Boring IV and Joel E. Short
School of Engineering and Applied Science
University of California, Los Angeles
Los Angeles, CA 90024
Telephone: (310) 206-8785
Email: waboring@cs.ucla.edu or jshort@cs.ucla.edu

Principal Investigators: Prof. Rajeev Jain and Prof. Leonard Kleinrock

*This work was supported by the U.S. Department of Justice/Federal Bureau of Investigation, ARPA/CSTO under Contract J-FBI-93-112 Computer Aided Design of High Performance Network Wireless Networked Systems

ABSTRACT

In this guide, we will look at the steps necessary for a programmer to add a networking protocol into WAMISNOS. This includes telling the compiler about the new protocol to be added, declaring the new protocols to the kernel and command line parser, how to connect multiple networking protocols together, kernel functions useful to networking protocol processes, and finally covers debugging tips.

Throughout the document, a logical link control protocol is used as an example. The protocol is used with TCP/IP and a wireless radio to achieve positive acknowledgments on the link level. The complete source code to the protocol is listed in the Appendix.

1. WAMISNOS overview.

WAMISNOS is a Network Operating System (NOS) enhanced for UCLA's Wireless Adaptive Mobile Information System (WAMIS) project. It is a multi-tasking operating system which runs on a PC-based computer running DOS. The WAMIS Network Operating System looks like an application in DOS. All of the kernel functions, applications, and networking protocols are compiled together into this application. Any protocol that is written for WAMISNOS is actually part of WAMISNOS itself, and is compiled as part of the.exe.

WAMISNOS is a command line based operating system, similar to DOS, but has multi-tasking capabilities and supports common network functionality. Once you start WAMISNOS, the user is able to start an application from the command line prompt. The user is also able to specify the initiation or destruction of any background process or network protocol. Each network protocol written for WAMISNOS can be its own process. However, a protocol is able to span multiple processes, or a process can contain multiple protocols. All the of scheduling and task switching functions are controlled by the WAMISNOS kernel. The WAMISNOS kernel provides various routines available to the network protocol programmer or to an applications programmer. More details on the available library functions can be found in the Appendix.

In this guide, we will look at the steps necessary for a programmer to add a networking protocol into WAMISNOS. These include telling the compiler about the new protocol, declaring the new protocols to the kernel and command line parser, how to connect multiple networking protocols together, kernel functions useful to protocol processes, and finally we give testing and debugging tips.

2. Step by step approach to writing a protocol as a process.

2.1. Overview

This section provides a beginner's overview of the steps involved in writing a protocol for WAMISNOS. In this document, we use the Logical Link Control (positive acknowledgment) protocol as an example. The complete source code for this protocol can be found in the appendix.

The first step in programming a network protocol for WAMISNOS is setting up the compiler for the file names so it can be merged in with the network operating system on the next build. To do this, additions (the file name which contains your protocol) need to be made to the makefile and library files.

Secondly, the operating system kernel needs to be told about the availability of the new protocol. This is done by modifying the *config.c* an *commands.h* files. In the *config.c* file, you add the process, or protocol, entry and exit calls. This is also where you let WAMISNOS know how to start your process at the command line prompt. In the *commands.h* file, you add the prototypes to your process entry and exit points so *config.c* knows where your functions are.

Third, special functions and functionality needs to be added to the network protocols. Process entry and exit points are the function calls that start-up and stop your protocol. They are called from *config.c* when you type "start <protocol>" at the command line prompt. The network protocol usually does process initialization and memory allocation which typically makes a call to initialize your protocol's structures in the process entry point. Lastly, the network protocol can start any child processes, if necessary. This is usually done in the process entry point.

2.2. Writing your protocol.

As mentioned earlier, we will be using a protocol for logical link control (positive acknowledgments), as an example. This protocol is contained in the llc.c and llc.h files. The purpose of this document is to describe how to write protocols for WAMISNOS, not to describe how Logical Link Control works.

2.2.1 Makefile additions.

You need to add your protocol to the WAMISNOS makefile, so it can be built when WAMISNOS is compiled. First of all you will need to create your protocol.c file that will hold your actual C code in it, and then create your .h, which contains any of your structures that you need, and also prototypes for all of your functions. After you have done this, you can add a couple of lines in the makefile that will tell the compiler how to create your object file, and how to link it into the rest of the WAMISNOS .exe file. Below is the line in the makefile that tells the compiler the dependencies of the *llc.obj* file. Any header file that is included in your protocol should be included as a dependency for you protocol.

[makefile]

```
llc.obj: llc.c llc.h global.h mbuf.h iface.h wamis.h pktdrvr.h config.h
commands.h

You also have to add your protocol.obj to the wamisobjs list to let the
compiler know its part of the WAMIS object.
WAMISOBJS= ansi.obj wamis.obj pwrctrl.obj speech.obj speechcmd.obj topo.obj \
     wamiscmd.obj speedser.obj speedcli.obj talkser.obj talkcli.obj \
     video.obj videocmd.obj video2.obj video2se.obj video2cl.obj \
     wmonitor.obj llc.obj llcmon.obj sync.obj cluster.obj\
     bercli.obj berser.obj elect.obj vgascrn.obj vtalk.obj
```

You will notice in this makefile that the WAMISNOS source code is broken down into modules. There are the kernel modules, TCP/IP modules, and also WAMIS modules. The wamis module (WAMISOBJS) contains all the routines added to the Network Operating System to support the WAMIS application and network protocol requirements.

2.2.2. Library file additions.

The makefile for WAMISNOS associates .obj files into libraries that are linked together to complete the executable code called *wamisnos.exe*. You need to add your protocol's object file to the WAMISNOS library declaration file *wamis.tl* so it can be linked into the WAMISNOS library when it is built into the *wamisnos.exe* file. This lets the compiler know how to build the .exe file from all of the object files that are created from all of the processes.

```
[wamis.tl]
...
+berser.obj &
+llc.obj &
+llcmon.obj &
...
```

2.2.3 Config.c additions

This is where you let WAMISNOS know how to start or stop your process at the command line prompt. You will usually start your process at the command line using the start command, and stop your process using the stop command. The calls to start your process can be added to the autoexec start-up file so it will be done automatically. The second argument to the start and stop commands is the process that you want to start and stop. For example if you wanted to start the LLC process you would type "*start llc*". You tell WAMISNOS what the start and stop commands are in the config.c file. In your declaration you tell the start/stop command what function call is associated with your start/stop entry/exit points. Then you specify how much stack space is required for your process, (1k is usually fine for most protocols). The <# params> is the number of parameters (*argc* & *argv*) that must be passed in after the start command. If the number of parameters passed in is less than this number, the command is treated as an error and the <description of command line options> is shown to the user so they will know the proper syntax to use this command. Both of these changes are done in a section of code in the *config.c* file listed below.

The format to declare how to start your process is:
"command name", <function>, <stack size>,<# params>,<description of command line options>

```
[Config.c]
...
#ifdef  SERVERS
/* "start" and "stop" subcommands */
static struct cmds DFAR Startcmds[] = {
...
   "llc",          llcstart,       1024, 0, NULLCHAR,
...
```

To declare how to stop your process, the config.c file is modified under the stop subcommands. The format is the same as the start subcommands:
"command name", <function>, <stack size>,<# params>,<description of command line options>

```
static struct cmds DFAR Stopcmds[] = {
...
   "llc",          llcstop,        0, 0, NULLCHAR,
...
```

When you call the start function, the start function creates a new process which your function runs inside of. Usually this function will loop forever and periodically give up CPU time to WAMISNOS to do other processing. However, this loop should check for a flag, which can be set by the stop function, so the process knows to stop looping. Once the loop fails (stop sets a finish flag), then the start function terminates and the process is killed automatically.

2.2.4 Commands.h additions

You add the prototypes to your process entry and exit points in here, so config.c knows how to call your process entry and exit functions. Any other functions which should be known in the WAMISNOS system (functions used outside your protocol) should also be declared here. For example, perhaps you would add a function which the trace function may call to dump a certain header or format of a packet. The example below shows the prototypes for the LLC protocol entry point and exit point.

[commands.h]

```
...
 /* In llc.c */
int llcstart __ARGS((int argc, char *argv[], void *p));
int llcstop  __ARGS((int argc, char *argv[], void *p));
...
```

2.2.5 Process entry and exit points.

These are the function calls that actually start and stop your protocol, as defined in *config.c* (step 2.2.3). The entry point is called from *config.c* when you type "start <protocol>" at the command line prompt. This is the point when the process is first launched. The routine that is called to stop your process, "stop <protocol>", is also defined in *config.c* (step 2.2.3). In the stop function, you should make sure that none of the routines in your process are running, then it should free up any memory that was allocated throughout the protocol.

[llc.c]

```
...
/* This is our main process for LLC.  It handles the initialization of our
   Data structures, then it starts a loop to handle our events. There are
   2 basic events that it can handle.  1) A packet has come in from the
   network, so check the Sequence #, and handle it. 2) Our most recently sent
   packet has timed out, so re-send it */
int
llcstart(argc,argv,p)
int argc;
char *argv[];
void *p;
{
...
}
   if (llc_init() == 1 )
      return 1;

   if(!(availmem() < Memthresh))
     /* Spawn a server */
     newproc("llctimer",2048,llctimer,0,NULL,NULL,0);
   while (llc_used)
   {  /* Wait for an event to signal us */
```

```
        ready=1;
        pwait(&LLC_Wakeup_Event);
        ready=0;
           llc_trash_pkt = 0;   /* Reset the trash packet global */

           if (llc_event == LLC_Get_pkt)
           { /* We recieved a packet from the Network */
              ...
           }

           else if (llc_event == LLC_Timeout)
           {    /* Our packet timed out */
              ...
           }

           else if (llc_event == LLC_Shutdown)
           {
               psignal(&LLC_Start_Timer,0);
               break;
           }
      }

return 0;
}

/* This function sets our global var to stop our LLC process. */
int
llcstop(argc, argv, p)
int argc;
char *argv[];
void *p;
{
        llc_used = 0;                /* Stop our loop we are done */
        llc_event = LLC_Shutdown;        /* we are shutting down! */
        psignal(&LLC_Wakeup_Event, 0);
        alert(pp,1);
        llc_cleanup();
        return 0;
}
```

2.2.6. Starting a child processes.

If your process needs to have child processes running then you need spawn a new process. For example, in the LLC protocol, a separate process for timing was needed because the timer couldn't be a part of the same process as what needed to be timed. So a new process was created called *llctimer()*. When *llcstart()* is called, LLC is started from the command line prompt with the command "start llc", llcstart spawns a new process called llctimer to handle all of the timing for llc. All processes are usually declared with the same parameters, but in the case of llctimer, void pointers were used for parameters since no parameters are needed. The normal parameters to a process are 1)int argc, 2)char *argv[], and 3)void *p. If you are spawning a process, that needs no parameters passed, just use the same example as in llctimer.

 The code below is called from the llcstart() function, and it starts a new process called llctimer, with a stack size of 2k. The parameters to newproc are:
“<Process name>”, <stack size>, <function call>, <argc>, <argv>, <pargv>,<freeargs>

pargv is commonly used as a session pointer. If *freeargs* are set, it should free the arguments from *argv* at termination.

[llc.c]

```
...
     if(!(availmem() < Memthresh))
   /* Spawn a server */
   newproc("llctimer",2048,llctimer,0,NULL,NULL,0);
...
```

Here is the declaration for llctimer:

[llc.c]

```
...
void
llctimer(a,b,c)
int a;
void *b;
void *c;
{
   int currnode = 0;
   int currpkt = 0;

   pp = Curproc;

   while (llc_used)
   {
      sigtimerok = 1;
      pwait(&LLC_Start_Timer);  /* Wait until we get called */
      sigtimerok=0;

      if (llc_event == LLC_Shutdown)
         break;

      currpkt = timerlist[nodetimer].pktnum; /* get pkt # we are timing */
      currnode = nodetimer; //store who we are timing on

      if (timervalue >= 1)
         pause(timervalue);
      else
         pause(Timeout);      /* Pause for the timeout value */

      timervalue = 0;

      if ((currnode==nodetimer)&&(currpkt == timerlist[nodetimer].pktnum))
      {
         llc_event = LLC_Timeout; /* Set the Event to Timeout event */
          while (!ready)
             pause(1);
```

```
            psignal(&LLC_Wakeup_Event,0); /* signal LLC to wake up */
        }
    }
}
```

2.2.7 What do you do with packets if your process isn't running.

In order to maintain the integrity of WAMISNOS when your process isn't running you need to declare a global variable that can be used to conditionalize calls to your routines so packets won't be sent to your process. In the LLC example, a global variable LLC_USED is created, and initialized to 0. When LLC is started LLC_USED is set to 1. The conditionalized calls will be explained in the next section for sending and receiving packets. If you have any child processes that you start, you should also add an if statement that will encompass all of the code inside of the process, so that if the global is set to 0, the process will quit. When your stop routine is called, it sets the global to zero, and in effect stops all of your other process from running and they exit out.

[llc.c]

```
int     llc_used = 0;           /* Global to let us know when to stop LLC */
```

In the example below, you can see that if the llc_used is set to 0, then the process will exit.

```
/* This process is what handles our timeout value for our packets */
/* this process check to see if our ttl has timed out.  If it does
            it signals llc_send to resend the packet */
void
llctimer(a,b,c)
int a;
void *b;
void *c;
{
   int currnode = 0;
   int currpkt = 0;

   pp = Curproc;

            while (llc_used)
            {
                        /* LLC TIMER CODE HERE */
            }
}
```

2.3. Sending and receiving packets.

2.3.1. What layer does your protocol fit, and do you need a new packet type?

Below is a basic picture of the structure of wamisnos. WAMIS applications make calls to TCP/IP either directly or through sockets. Then IP passes the packet down to the WAMISNOS network layer. This is where most of the protocols are written in wamisnos, to take control of the link level. If you were to write a protocol to handle virtual circuits, one would write an application that would make calls directly to a VC protocol, instead of the TCP/IP protocols, then the VC should pass the packet on down to the WAMIS protocols area in the stack the same way that IP does. The next picture on the next page shows how the packets are passed from layer to layer, and what functions are called.

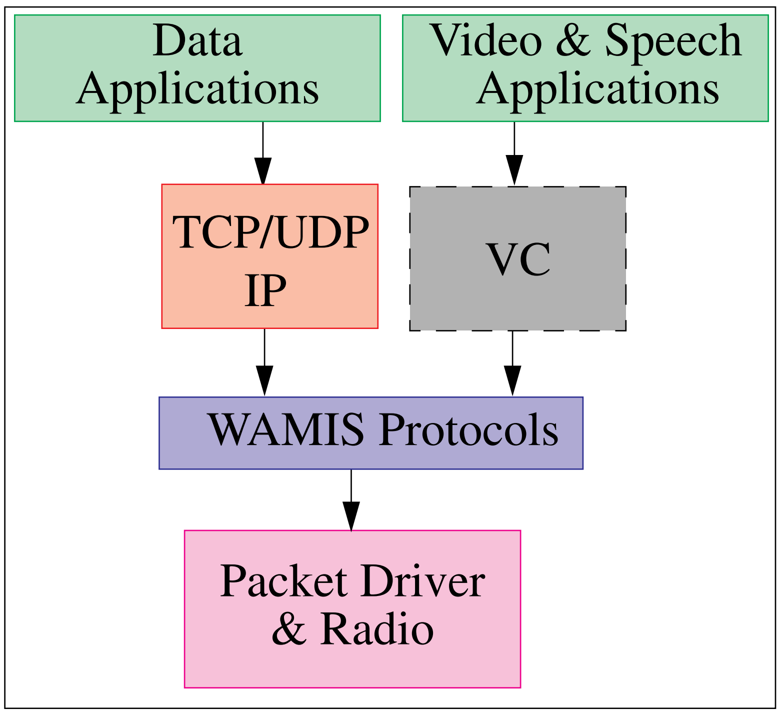

Figure 1 WAMISNOS Network Layers

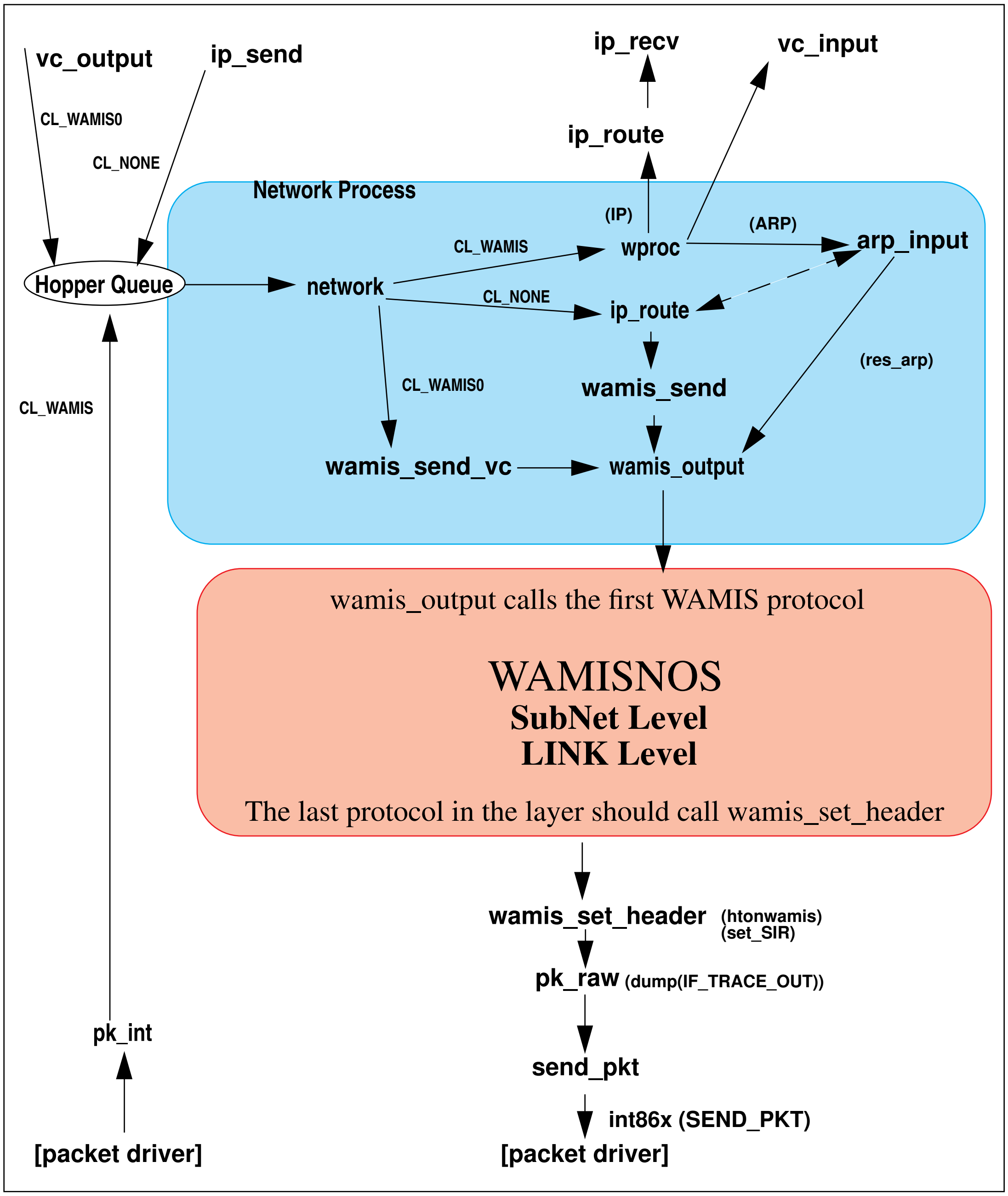

Figure 2 WAMISNOS Network Protocol Stack

2.3.2 Inserting the hooks into the WAMISNOS kernel.

In order for your protocol to do work on packets, WAMISNOS has to pass the incoming, or outgoing packets to you. This is the place when you need to make a conditionalized call to you input packet routing. If your protocol isn't running then you should not call your routine, and just let the next protocol on the stack take the packet. If your protocol is running, you should make a call to your input routine, then when you are done with the packet, pass it on to the next layer if necessary.

Below is how the outgoing packets are passed to your protocol. The packets would be typically coming from an application, and you catch them before they go out to the network. Inside the routine that handles packets sent to it by the upper layers, it should make sure the protocol is running, if it isn't it should just pass the packet down to the next protocol, leaving the packet untouched.

[wamis.c]

```
/* Send a packet to the Clustering alg, then to the
   Link level control. */
int
wamis_output(iface,dest,source,type,data)
struct iface *iface;     /* Pointer to interface control block */
char *dest;              /* Destination WAMIS address */
char *source;            /* Source WAMIS address */
int16 type;              /* Type field */
struct mbuf *data;       /* Data field */
{
      #ifdef HDEBUG
        printf("\nWAMIS_OUTPUT:Send a packet to node %d\n", dest[2]);
        fflush(stdout);
      #endif

   /* Send the packet to the Link Level Control */
   llc_q_pkt(iface,dest,source,type,data);

   return 0;
} /* wamis_output() */

/* Put the packet in our send Q. */
int
llc_q_pkt(iface,dest,source,type,ptr)
struct iface *iface;     /* Pointer to interface control block */
char *dest;              /* Destination WAMIS address */
char *source;            /* Source WAMIS address */
int16 type;              /* Type field */
struct mbuf *ptr;        /* Pointer to the packet */
{
             struct ph bph; /* our header information  */
             int node_id = 0;

/* Make sure we aren't trying to send after llc has closed down */
             if (llc_used && (type == IP_TYPE))
             {
              ...
              /* Process packet*/
              ...
               return 0;
             }
             else
               {
                 wamis_set_header(iface,dest,source,type,ptr);
```

```
            }
        return 0;
}
```

Below is the code to pass incoming packets to your protocol in the *wproc* function inside of the [wamis.c] file. These packets typically are coming in from the network, and are on their way back up to the upper layers in the protocol stack. You can see the first thing it does is to make sure that the protocol is actually running, if it is then it should pass the packet into the protocol.

```
[wamis.c]
/* Wake up LLC, and have it check the packet seq #. */
if (llc_used)  /* if we are using LLC */
  if (llc_wakeup(&hdr) == 1)
   {
     /* The packet is out of sequence so trash it, and get out */
      d_badseq++;
      free_p(bp);
      return;
   }
```

2.3.3 Manipulating packets

2.3.3.1. What is an mbuf?

An mbuf (memory buffer) is the structure in which packets or blocks of data are kept. An mbuf can be viewed as simply a memory structure that standardizes memory allocations and de-allocations to the kernel. A packet is one or more mbufs. Mbufs can be linked together (via the *struct mbuf *next*) to make up a packet which is larger then what can be stored in one mbuf. Anytime a packet of data is received, it is put into an mbuf. The programmer should be aware that the packet can span multiple mbufs so the *mbuf->next* should be checked to see if the packet continues on to the next mbuf.

[mbuf.h]

```
/* Basic message buffer structure */
struct mbuf {
   struct mbuf *next;      /* Links mbufs belonging to single packets */
   struct mbuf *anext;     /* Links packets on queues */
   int16 size;             /* Size of associated data buffer */
   int refcnt;             /* Reference count */
   struct mbuf *dup;       /* Pointer to duplicated mbuf */
   char *data;             /* Active working pointers */
   int16 cnt;              /* # bytes being used in mbuf (max is 'size') */
};
```

The parameter mbuf->size is the size of the mbuf as it was created by the *ambufw()* or a similar function. The mbuf->cnt contains how many bytes are used starting at mbuf->data. The mbuf->cnt should be set anytime you add or remove data in the mbuf.

2.3.3.2. Creating and Deleting packets.

This shows the code to make a call to *ambufw()*, which allocates an mbuf with a data size of 1K. Any time you create a packet (such as from an application) you should try and leave extra room in the mbuf so the header can be added on. If you don't make the original mbuf large enough, the multiple mbufs will have to be linked together.

[llc.c]

struct mbuf *ack_pkt_ptr; /* pointer to the ack packet's mem */

```
    ack_pkt_ptr = ambufw(1024);   /* Allocate memory for our packet */
    ack_pkt_ptr->cnt = 0;      /* set the amount used to 0 */
```

The next line shows how the memory used by an mbuf is given back to the heap.

```
  struct mbuf *pkptr;  /* pointer to a packet */
    free_p(pkptr);        /* Free the memory allocated by it */
```

2.3.3.3. Copying packets.

You can make a copy of a packet by making a call to the function dup_p() in [mbuf.c]
[mbuf.c]

```
/* Duplicate first 'cnt' bytes of packet starting at 'offset'.
 * This is done without copying data; only the headers are duplicated,
 * but without data segments of their own. The pointers are set up to
 * share the data segments of the original copy. The return pointer is
 * passed back through the first argument, and the return value is the
 * number of bytes actually duplicated.
 */
int16
dup_p(hp,bp,offset,cnt)
struct mbuf **hp;
register struct mbuf *bp;
register int16 offset;
register int16 cnt;
{
...
}
```

2.3.3.4. Queueing packets.

You can queue and de-queue packets using the functions in mbuf.c.
[llc.c]

```
struct mbuf *ptr;               /* Pointer to the packet */
struct mbuf *llc_pkt_q;         /* This is our pointer to our Queue */

  enqueue(&llc_pkt_q, ptr);     /* Queue our packet */
  ptr = dequeue(&llc_pkt_q);    /* take the packet off of the Queue */
```

2.3.3.5. Putting information into a packet.

You can place new information into a packet, or mbuf, by pushing information into the data portion of the mbuf (updating *mbuf->cnt*). If *mbuf->cnt* becomes larger than *mbuf->size*, then pushdown will increase *mbuf->size*. First create new space in the mbuf by calling pushdown, then use memcpy to copy the information into the new space.

```
/* create space in the data portion of the packet for our header */
ptr=pushdown(ptr,sizeof(bph));

/* save our packet header info in the data part of the packet */
memcpy (ptr->data,(char *)&bph, sizeof(bph));
```

2.3.3.6. Removing information from a packet.

You can take the information off of a packet, or mbuf, by basically doing the reverse of putting the information on. Just make a call to pullup.

```
/*remove header from packet and put in struct ph */
pullup(&pkptr,(char *)&bph, sizeof(struct ph));
```

2.4. Giving up the CPU to WAMISNOS.

2.4.1 Pausing

The pause command is used to pause a process for a specified amount of time, in milliseconds. While your process is paused, it releases time to WAMISNOS to handle O.S. functions, and gives time up to other processes that need time on the CPU. This is similar to the sleep command except that a sleep command would put the whole WAMISNOS to sleep where the pause command only pauses the current process so other processes in WAMISNOS can run. For example, to have your current process pause for 1 second, you would write:

```
pause(1000);
```

2.4.2 Wait and signal semaphores

Pwait(), and *psignal*() operate on the same premiss as the classical wait and signal semaphore. *Pwait()* halts a process conditioned on a semaphore, and then gives up the until it is notified vie the semaphore by *psignal()*. For example, at some point in your protocol, you might have a state at which you no longer need the CPU. You could be waiting for another packet to come in, or waiting for a timer to time out. Use *pwait()* to give up time back to WAMISNOS to process more incoming and outgoing packets, then when you need to wake your process back up, use *psignal()*. The parameter passed to *pwait()* and *psignal()* is the semaphore (an integer variable). See section 2.2.5 for an example of the pwait and psignal functions.

3. Debugging your protocol

3.1 How to get information on your variables.

Since a source code level debugger doesn't work while WAMISNOS is running, you have to use the age old printf debugging technique. Make sure you use #ifdef statements so that you can keep your debugging code in after you are done debugging, for use at a later date. There is just one caveat with debugging this way in WAMISNOS. In order to ensure that all of your debugging information is printed to the screen when the printf is called you need to make a call to fflush. When you make a call to printf the string that you want to get printed get's put into a buffer to be drawn on the string the next time the buffer is flushed. So just make a call to fflush(stdout) after all of your printf's.

There is also a function called tprintf which is available. This function accepts parameters similar to printf but queues all output for the correct stream (output port). This means that the tprintf will only display on the screen if the processes which calls tprintf is in the foreground. If the process is in the background and a call to tprintf is made, the data is queued for the next time that process is brought into the foreground. The tprintf function will not wait for the process to be brought to the foreground but send the data to the output port (queue.)

Appendix

A. Memory Buffer Functions (mbuf.c, mbuf.h)

#include “mbuf.h”
The Mbuf struct is declared as:

```
/* Basic message buffer structure */
struct mbuf {
            struct mbuf *next;   /* Links mbufs belonging to single packets */
            struct mbuf *anext;     /* Links packets on queues */
            int16 size;             /* Size of associated data buffer */
            int refcnt;             /* Reference count */
            struct mbuf *dup;       /* Pointer to duplicated mbuf */
            char *data;             /* Active working pointers */
            int16 cnt;
};
#define NULLBUF (struct mbuf *)0
#define NULLBUFP (struct mbuf **)0
```

alloc_mbuf: Allocate mbuf with associated buffer of 'size' bytes. If interrupts are enabled, use the regular heap. If they're off, use the special interrupt buffer pool.
Declaration:
`struct mbuf *alloc_mbuf(register int16 size)`

ambufw: Allocate mbuf, waiting if memory is unavailable
Declaration:
`struct mbuf *ambufw(int16 size)`

free_mbuf: Decrement the reference pointer in an mbuf. If it goes to zero, free all resources associated with mbuf.Return pointer to next mbuf in packet chain.
Declaration:
`struct mbuf *free_mbuf(register struct mbuf *bp)`

free_p: Free packet (a chain of mbufs). Return pointer to next packet on queue, if any
Declaration:
`struct mbuf *free_p(register struct mbuf *bp)`

Free_q: Free entire queue of packets (of mbufs)
Declaration:
`void free_q(struct mbuf **q)`

Len_p: Count up the total number of bytes in a packet
Declaration:
`int16 len_p(register struct mbuf *bp)`

len_q: Count up the number of packets in a queue
Declaration:
```
int16 len_q(register struct mbuf *bp)
```

trim_mbuf: Trim mbuf to specified length by lopping off end
Declaraion:
```
void trim_mbuf(struct mbuf **bpp, int16 length)
```

dup_p: Duplicate first 'cnt' bytes of packet starting at 'offset'. This is done without copying data; only the headers are duplicated, but without data segments of their own. The pointers are set up to share the data segments of the original copy. The return pointer is passed back through the first argument, and the return value is the number of bytes actually duplicated.
Declaration:
```
int16 dup_p(struct mbuf **hp, register struct mbuf *bp,
            register int16 offset, register int16 cnt)
```

copy_p: Copy first 'cnt' bytes of packet into a new, single mbuf
Declaration:
```
struct mbuf *copy_p(register struct mbuf *bp, register int16 cnt)
```

pullup: Copy and delete "cnt" bytes from beginning of packet. Return number of bytes actually pulled off.
Declaration:
```
int16 pullup(struct mbuf **bph,char *buf,int16 cnt)
```

append: Append mbuf to end of mbuf chain */
Declaration:
```
void append(struct mbuf **bph,struct mbuf *bp)
```

pushdown: Insert specified amount of contiguous new space at the beginning of an mbuf chain. If enough space is available in the first mbuf, no new space is allocated. Otherwise a mbuf of the appropriate size is allocated and tacked on the front of the chain. This operation is the logical inverse of pullup(), hence the name.
Declaration:
```
struct mbuf *pushdown(register struct mbuf *bp, int16 size)
```

enqueue: Append packet to end of packet queue.
Declaration:
```
void enqueue(struct mbuf **q,struct mbuf *bp)
```

dequeue: Unlink a packet from the head of the queue.
Declaration:
struct mbuf *dequeue(register struct mbuf **q)

qdata: Copy user data into an mbuf.
declaration:
struct mbuf *qdata(char *data, int16 cnt)

dqdata: Copy mbuf data into user buffer.
Declaraiont:
int16 dqdata(struct mbuf *bp, char *buf,unsigned cnt)

pull32: Pull a 32-bit integer in host order from buffer in network byte order.On error, return 0. Note that this is indistinguishable from a normal return.
Declaration:
int32 pull32(struct mbuf **bpp)

pull16: Pull a 16-bit integer in host order from buffer in network byte order. Return -1 on error
Declaration:
long pull16(struct mbuf **bpp)

pullchar: Pull single character from mbuf.
Declaration:
int pullchar(struct mbuf **bpp)

write_p: I think this writes the mbuf to a file.
Declaration:
int write_p(FILE *fp, struct mbuf *bp)

mbuf_crunch: Reclaim unused space in a mbuf chain. If the argument is a chain of mbufs and/or it appears to have wasted space, copy it to a single new mbuf and free the old mbuf(s). But refuse to move mbufs that merely reference other mbufs, or that have other headers referencing them. Be extremely careful that there aren't any other pointers to (or into) this mbuf, since we have no way of detecting them here. This function is meant to be called only when free memory is in short supply.
Declaration:
void mbuf_crunch(struct mbuf **bpp)

B. Kernel Functions (kernel.c kernel.h)

```
#include "proc.h"
/* Kernel process control block */
struct proc {
            struct proc *prev;/* Process table pointers */
            struct proc *next;

            jmp_buf env;            /* Process state */
            char i_state;           /* Process interrupt state */

            unsigned short state;
#define     READY       0
#define     WAITING     1
#define     SUSPEND     2
            void *event;            /* Wait event */
            int16 *stack;           /* Process stack */
            unsigned stksize;/* Size of same */
            char *name;             /* Arbitrary user-assigned name */
            int retval;             /* Return value from next pwait() */
            struct timer alarm;/* Alarm clock timer */
            struct mbuf *outbuf;/* Terminal output buffer */
            int input;              /* standard input socket */
            int output;             /* standard output socket */
            int iarg;               /* Copy of iarg */
            void *parg1;            /* Copy of parg1 */
            void *parg2;            /* Copy of parg2 */
            int freeargs;           /* Free args on termination if set */
};
```

newproc: Create a new, ready process and return pointer to descriptor. The general registers are not initialized, but optional args are pushed on the stack so they can be seen by a C function.

Declaration:

```
struct proc *newproc(name,stksize,pc,iarg,parg1,parg2,freeargs)
   char *name;               Arbitrary user-assigned name string
   unsigned int stksize;     Stack size in words to allocate
   void (*pc)();             Initial execution address
   int iarg;                 Integer argument (argc)
   void *parg1;             Generic pointer argument #1 (argv)
   void *parg2;    Generic pointer argument #2 (session ptr)
   int freeargs;  If set, free arg list on parg1 at termination
```

killproc: Free resources allocated to specified process. If a process wants to kill itself, the reaper is called to do the dirty work. This avoids some messy situations that would otherwise occur, like freeing your own stack.

Declaration:

```
void killproc(register struct proc *pp)
```

killself: Terminate current process by sending a request to the killer process. Automatically called when a process function returns. Does not return.
Declaration:
void killself(void)

killer: Process used by processes that want to kill themselves.
Declaration:
void killer(int i,void *v1,void *v2)

suspend: Inhibit a process from running.
Declaration:
void suspend(struct proc *pp)

resume: Restart suspended process.
Declaration:
void resume(struct proc *pp)

alert: Wakeup waiting process, regardless of event it's waiting for. The process will see a return value of "val" from its pwait() call.
Declaration:
void alert(struct proc *pp, int val)

pwait: Post a wait on a specified event and give up the CPU until it happens. The null event is special: it means "I don't want to block on an event, but let somebody else run for a while". It can also mean that the present process is terminating; in this case the wait never returns.

Pwait() returns 0 if the event was signaled; otherwise it returns the arg in an alert() call. Pwait must not be called from interrupt level.

Note that pwait can run with interrupts enabled even though it examines a few global variables that can be modified by psignal at interrupt time. These *seem* safe.
Declaration:
int pwait(void *event)

psignal: processes will see a return value of 0 from pwait(). Note that they don't actually get control until we explicitly give up the CPU ourselves through a pwait(). Psignal may be called from interrupt level. It returns the number of processes that were woken up.
Declaration:
```
int psignal(event,n)
   void *event;    /* Event to signal */
```

```
   int n;           /* Max number of processes to wake up */
```

chname: Rename a process. Moves *newname value into pp->name.
Declaration:

```
void chname(struct proc *pp, char *newname)
```

delproc: Remove a process entry from the appropriate table.
declaration:

```
static void delproc(entry)
register struct proc *entry;    /* Pointer to entry */
```

addproc: Append proc entry to end of appropriate list.
Declaration:

```
static void addproc(entry)
register struct proc *entry;    /* Pointer to entry */
```

C. Timer Specific Functions (timer.c timer.h)

```
#include "timer.h"
struct timer {
     struct timer *next;              /* Linked-list pointer */
     int32 duration;                  /* Duration of timer, in ticks */
     int32 expiration;                /* Clock time at expiration */
     void (*func) __ARGS((void *));   /* Function to call at expiration */
     void *arg;                       /* Arg to pass function */
     char state;                      /* Timer state */
#define     TIMER_STOP  0
#define     TIMER_RUN   1
#define     TIMER_EXPIRE2
};
```

start_timer: Start a timer.
Declaration:
`void start_timer(struct timer *t)`

stop_timer: Stop a timer.
Declaration:
`void stop_timer(struct timer *timer)`

read_timer: Return milliseconds remaining on this timer.
Declaration:
`int32 read_timer(struct timer *t)`

set_timer: Sets the timer value in ticks.
Declaration:
`void set_timer(struct timer *t, int32 interval)`

pause: Delay process for specified number of milliseconds.Normally returns 0; returns -1 if aborted by alarm.
Declaration:
`int pause(int32 ms)`

t_alarm: I don't know about this one.
Declaraiont:
`static void t_alarm(void *x)`

alarm: Send signal to current proc. after specified milliseconds.
Declaration:
`void alarm(int32 ms)`

tformat: Convert time count in seconds to printable days:hr:min:sec format.
Declaration:
`char *tformat(int32 t)`

D. WAMIS Specific Functions (wamis.c wamis.h)

```
#include "wamis.h"
```
setdcode: Set the default code to transmit on
Declaration:
`int setdcode(char newcode)`

setdpower: Set the default power to transmit on.
Declaration:
`int setdpower(char newpower)`

setddspread: Set the default data spreading factor to transmit with.
Declaration:
`int setddspread(char newspread)`

setdspread: Set the default Speech spreading factor to transmit with.
Declaration:
`int setdsspread(char newspread)`

setdvspread: Set the default speech spreading factor to transmit with.
Declaration:
`int setdvspread(char newspread)`

setdrcode: Set the default code to receive on.
Declaration:
`int setdrcode(char newrcode)`

setdspread: Set the default spreading factor to recieve on.
Declaration:
`int setdrspread(char newspread)`

htonwamis: Add on WAMIS header.
Declaration:
`struct mbuf *htonwamis(struct wamishdr *wamishdr,struct mbuf *bp)`

ntohwamis: Remove WAMIS header.
Declaration:
`int ntohwamis(struct wamishdr *wamishdr,struct mbuf **bpp)`

wamis_get_node_id: This function returns the id of the node we are running on. It gets the information from the ip address. 128.97.93.3 would be node 3, because the last digit is a 3.

```
Declaration:
int wamis_get_node_id()
```

wamis_send_vc: Process Queued WAMIS Output packets. This would be of type CL_WAMISO from Hopper.

```
Declaration:
int wamis_send_vc(struct iface *iface,struct mbuf *bp;
```

wproc: Process Queued WAMIS Input packets

```
Declaration:
void wproc(struct iface *iface,struct mbuf *bp)
```

wamis_output: Send a packet to the Clustering alg, then to the Link level control.

```
Declaration:
int wamis_output(iface,dest,source,type,data)
   struct iface *iface;  /* Pointer to interface control block */
   char *dest;           /* Destination WAMIS address */
   char *source;         /* Source WAMIS address */
   int16 type;           /* Type field */
   struct mbuf *data;    /* Data field */
```

wamis_set_header: Send a packet with WAMIS header.

```
Declaration:
int wamis_set_header(iface,dest,source,type,data)
   struct iface *iface; /* Pointer to interface control block */
   char *dest;          /* Destination WAMIS address */
   char *source;           /* Source WAMIS address */
   int16 type;             /* Type field */
   struct mbuf *data;      /* Data field */
```

wamis_send: Send an IP datagram on WAMIS.

```
Declaration:
int wamis_send(bp,iface,gateway,prec,del,tput,rel)
   struct mbuf *bp;      /* Buffer to send */
   struct iface *iface;  /* Pointer to interface control block */
   int32 gateway;        /* IP address of next hop */
   int prec;
   int del;
   int tput;
   int rel;
```

E. Interface Functions (iface.c iface.h)

#include "iface.h"

if_lookup:Given the ascii name of an interface, return a pointer to the structure,or NULLIF if it doesn't exist.
Declaration:
struct iface *if_lookup(char *name)

ismyaddr:Return iface pointer if 'addr' belongs to one of our interfaces,NULLIF otherwise. This is used to tell if an incoming IP datagram is for us, or if it has to be routed.
Declaration:
struct iface *ismyaddr(int32 addr)

ifmtu: Set interface Maximum Transmission Unit
Declaration:
static intifmtu(int argc,char *argv[],void *p)

ifdesc:give a little description for each interface
Declaration:
static int ifdescr(int argc, char *argv[],void *p)

ifipaddr: Set interface IP address. argv holds the address, and p holds the iface pointer.
Declaration:
static int ifipaddr(int argc, char *argv[],void *p)

F. LLC.H

```
/* File: LLC.H
 * Created By: Walt Boring IV @ UCLA
 */

#ifndef _LLC_H
#define _LLC_H

/* Generic Ethernet constants and templates */
#ifndef _GLOBAL_H
#include "global.h"
#endif

#ifndef _MBUF_H
#include "mbuf.h"
#endif

#ifndef _IFACE_H
#include "iface.h"
#endif

#ifndef _ENET_H
#include "enet.h"
#endif

extern struct mbuf *llc_pkt_q;

/* Link Level Control Functions */
int llc_q_pkt __ARGS((struct iface *iface,char dest[],char source[],int16
type, struct mbuf *data));

int llc_seq_num(int16 type, int nodeid);
int llc_wakeup(struct wamishdr *hdr);
int llc_handle_ack(int packnum, int nodeid);
int llc_ack_pkt(int pktnum, int nodeid);
int llc_packet_resend(int pktnum);
int llcstart __ARGS((int argc, char *argv[], void *p));
int llcstop  __ARGS((int argc, char *argv[], void *p));
void llc_cleanup(void);
int llc_init(void);
void llctimer(int a, void *b, void *c);
int llc_pack_num(int16 type, int nodeid);
int llc_snd_pkt(int nodeid);
void init_nodes(void);
int  get_node_id(int32 addr);
void addtolist(int nodenum, int packnum);
void removefromlist (int nodenum, int packnum);
void checklist(void);

void llcq_timer(int nodenum, int packnum);
void llc_stat(void);
```

```
/* LLC Kernel Globals */
extern int LLC_Wakeup_Event;  /*1000   start our wait state */
extern int LLC_Start_Timer;   /*1001    Start our llc_timer process */
extern int LLC_Done;          /*1002    Let us know when we are done w/
current packet */
#define LLC_Timeout     2000    /* Packet Timout event */
#define LLC_Get_pkt     2001    /* Network recievec a packet event */
#define LLC_Shutdown    2002    /* Shutdown our LLC stuff */


extern int llc_used;            /* So wamis can access this */
extern int llc_seq_num_out;

/* LLC Globals */
#define Timeout 2500  /* set our timout to 1500ms or 1.5 seconds */
#endif /* Link Level header */

/* Any LLC Structs */
struct ph
{
   struct iface *iface;
   char *dest;
   char *source;
   int16   type;
   struct mbuf *data;
};



#define numnodes 4  //The number of nodes on our network

struct node
{
  struct mbuf *pkt_q;  //Holds the pointer to our packet Q
  int seq_num_in;      //Holds the sequence # of the incomming pkt
  int seq_num_out;     //Holds the sequence # of the outgoing pkt;
  int ack_num;         //Holds the last recieved packet # for nod x/
};

typedef struct node node;


//This is our structure for our timing object
struct timernode
{
  long timestamp;    //Holds the time at which this entry was Queued
  int timetowait;    //Holds the time it wanted to wait for it's timeout
  int nodeid;        //Holds the node this packet is destined for
  int pktnum;        //Holds the sequence # of the packet we are timing
};
typedef struct timernode timernode;
```

G. LLC.C

```
/* This is the Link Level Control Protocol handlers */

#undef FDEBUG 1// Function entry/exit point debugging
#undef PDEBUG 1// Packet debugging information
#undef TDEBUG 1// Timing debugging information
#undef MDEBUG 1// Memory debugging information
#undef WDEBUG 1// generic debugging information
/* Link Level Control specific functions
 * File: LLC.C
 * Written by Walt Boring IV
 */

#include <stdio.h>
#include <time.h>
#include "global.h"
#include "mbuf.h"
#include "iface.h"
#include "arp.h"
#include "ip.h"
#include "enet.h"
#include "wamis.h"
#include "llc.h"
#include "timer.h"
#include "speech.h"
#include "video.h"
#include "internet.h"
#include "trace.h"

/* Variables */

struct mbuf *llc_pkt_q;      /* This is our pointer to our Queue */
int     llc_used = 0;        /* Global to let us know when to stop LLC */
int     llc_seq_num_in = 0;  /* Holds our current sequence we are expecting */
int     llc_seq_num_out = 0;    /* Holds the sequence number of out packets */
int     llc_event;              /* Holds the event to be processed */
struct  wamishdr *in_pkt_hdr;   /* Pointer to the incomming packet header */

/* don't trash it if it's 0, trash the packet if it's 1 */
int     llc_trash_pkt;
struct  proc *pp; /* Use this to keep track of our Timer proc */
int sigtimerok;
intack_num; /* Holds the # of the packet to ack */
intllc_cntrl=0; /* used to tell us to rest our counters */

int LLC_Wakeup_Event;   /* start our wait state */
int LLC_Start_Timer;    /* Start our llc_timer process */
int LLC_Done;           /* Let us know when we are done w/ current packet */

node nodelist[numnodes + 1];        /* Our list of nodes */
timernode timerlist[numnodes + 1];  /* our timer list */
int timerprec[numnodes + 1];  /* precedence list (Q)to poll our timer list */
int nodetimer = 0;
```

```
int pkttimer = 0;
long timervalue=0;
int ready = 0;

/* This process sets up our event type to be processed from wproc.  Then
   it signals our process to wake up to handle the event.  Then it returns
   the result of the event processing. */

int
llc_wakeup(hdr)
struct wamishdr *hdr;
{
    /* set the event type to handle the incomming packet */
   llc_event = LLC_Get_pkt;

    /* set our header pointer to the incomming packets header */
   in_pkt_hdr = hdr;
   psignal(&LLC_Wakeup_Event,0);  /* Signal our process to wake up */
   pwait(&LLC_Done);              /* Wait until it's done */
   return llc_trash_pkt;          /* Pass back the result */
}

/* This is our main process for LLC.  It handles the initialization of our
   Data structures, then it starts a loop to handle our events. There are
   2 basic events that it can handle.  1) A packet has come in from the
   network, so check the Sequence #, and handle it. 2) Our most recently sent
   packet has timed out, so resend it */
int
llcstart(argc,argv,p)
int argc;
char *argv[];
void *p;
{
   if (llc_init() == 1 )
     return 1;

   if(!(availmem() < Memthresh))
      /* Spawn a server */
      newproc("llctimer",2048,llctimer,0,NULL,NULL,0);


   while (llc_used)
      {
        /* Wait for an event to signal us */
        ready=1;
        pwait(&LLC_Wakeup_Event);
        ready=0;
        llc_trash_pkt = 0;   /* Reset the trash packet global */

        /* We recieved a packet from the Network */
        if (llc_event == LLC_Get_pkt)
         {

          if (in_pkt_hdr->type == ARP_TYPE)
```

```
            {
              llc_trash_pkt = 0;  /* we don't care about this packet */
              #ifdef PDEBUG
                printf("LLCSTART:Got an ARP \n");
              #endif
            }

           else if (in_pkt_hdr->type == LLC_TYPE) /* this is an explicit ACK */
            {
              llc_trash_pkt = 0;        /* tell w_proc to delete the packet */
              #ifdef PDEBUG
                 printf("LLCSTART:We GOT AN ACK packet \n");
              #endif
              /* handle the Ack # */
              llc_handle_ack(in_pkt_hdr->pack, in_pkt_hdr->source[2]);
            }
          else
          {
            llc_trash_pkt = 0;          /* The seq # is correct */

            #ifdef PDEBUG
              printf("LLCSTART:Got an IP Packet #%d from Node%d\n", in_pkt_hdr-
>pktnum, in_pkt_hdr->source[2]);
            #endif

           /* send an ack to the received packet */
           llc_ack_pkt(in_pkt_hdr->pktnum, in_pkt_hdr->source[2]);
           if (nodelist[in_pkt_hdr->source[2]].seq_num_in == in_pkt_hdr-
>pktnum)
              /* update the current input seq # */
              nodelist[in_pkt_hdr->source[2]].seq_num_in++;
           else
             {
               llc_trash_pkt = 1; /* We already have this packet */
               #ifdef WDEBUG
                 printf("LLCSTART:We already got this packet\n");
               #endif
             }
           }
           psignal(&LLC_Done,0);
        }

     else if (llc_event == LLC_Timeout)     /* Our packet timed out */
       {
         #ifdef PDEBUG
          printf("LLCSTART:Resend a packet for node %d\n", timerprec[1]);
         #endif
         llc_snd_pkt(timerprec[1]);    /* resend it */
       }
     else if (llc_event == LLC_Shutdown)
       {
         psignal(&LLC_Start_Timer,0);
         break;
       }
```

```
    }

return 0;
}

/* This function checks the ack # and makes sure it is the last sent packet.
   If it isn't then it needs to send the packet again. If the ack # was the
   # of the last sent packet, then it removes the packet off the Q. */

int
llc_handle_ack(packnum, nodeid)
int packnum;
int nodeid;
{
  struct mbuf *pkptr=NULL;  /* pointer to a packet */
  struct ph *bph;           /* pointer to our header */
  int done = 0;             /* flag to get out of our loop */

    #ifdef FDEBUG
       printf("LLC_HANDLE_ACK: Node #%d :packnum %d: seq_num_out %d\n",
nodeid, packnum, nodelist[nodeid].seq_num_out);
    #endif

  /* check to see if the packnum is the same as the last packet sent. */
      if (packnum == nodelist[nodeid].seq_num_out)
        {
          /* ok we are in sequence, so just delete the packet from the Q */
          #ifdef PDEBUG
            printf("LLC_HANDLE_ACK:Got an ACK for Node %d's Packet #%d\n"
,nodeid, packnum);
          #endif

          pkptr = dequeue(&nodelist[nodeid].pkt_q);
          free_p(pkptr);        /* Free the memory allocated by it */
          nodelist[nodeid].seq_num_out++;

          #ifdef TDEBUG
            printf("LLC_HANDLE_ACK:alert our timer to wake up now \n");
          #endif

          removefromlist(nodeid, packnum);

          if (in_pkt_hdr->pktnum == 1)
            nodelist[nodeid].seq_num_out = 1;
          llc_snd_pkt(nodeid);       /* Now send the next packet in the Q */
        }

    return 0;
}



/* This function acknowledges to the sender that we have recieved the packet
   that we wanted, and it also sends an ack of the last packet received if
```

```
   we got a packet number greater than the number that we expected. */
int
llc_ack_pkt(pktnum, nodeid)
int pktnum;
int nodeid;
{
  struct mbuf *ack_pkt_ptr;  /* pointer to the ack packet's mem */
  struct iface *iface;       /* pointer to our wamis interface port */

   #ifdef FDEBUG
      printf("LLC_ACK_PKT:Acking a packet %d\n",pktnum);
   #endif

   ack_pkt_ptr = ambufw(1);      /* Allocate memory for our packet */
   ack_pkt_ptr->cnt = 1;         /* set the size var to 100 */
   iface = if_lookup("w0");      /* get a ptr to the local wamis port */

    nodelist[nodeid].ack_num = pktnum;
    #ifdef PDEBUG
      printf("LLC_ACK_PKT:Explicit Ack packet #%d::%d\n", pktnum,
nodelist[nodeid].seq_num_in);
    #endif

    if(pktnum == 254)
     {
      llc_cntrl = 1;    /* tell the sender to reset the stats */
      nodelist[nodeid].seq_num_in = 0;
     }
    else llc_cntrl = 0;

    wamis_set_header(iface,in_pkt_hdr->source,iface->hwaddr,
LLC_TYPE,ack_pkt_ptr);

    return 0;
}

/* This function returns the number of the last received packet */
/* If the Packet type is a non ACK packet then return 0 */
int
llc_pack_num(type,nodeid)
int16 type;
int nodeid;
{

   if (type == LLC_TYPE)
     {
       #ifdef FDEBUG
          printf("LLC_PACK_NUM: return Ack #%d for Node #%d\n",
nodelist[nodeid].ack_num, nodeid);
       #endif
       /* We are acking a packet, so return the pkt # */
       return nodelist[nodeid].ack_num;
     }
   else
```

```
      {
        #ifdef FDEBUG
          printf("LLC_PACK_NUM: return 0.  WE aren't acking in this packet\n");
        #endif
        return 0;  /* we aren't acking a packet */
      }
}


/* This function returns the current Sequence number for outgoing packets. */
/* if the packet type is an explicit ACK, then put it 0, for the seq # */
int
llc_seq_num(type, nodeid)
int16 type;
int nodeid;
{
   if (type == LLC_TYPE)
      {
        #ifdef FDEBUG
          printf("LLC_SEQ_NUM: return llc_cntrl\n");
        #endif
        return llc_cntrl;     /* The packet is an Explicit Ack */
      }
   else
      {
        #ifdef FDEBUG
          printf("LLC_SEQ_NUM: return the seq num #%d for Node
%d\n",nodelist[nodeid].seq_num_out, nodeid);
        #endif
       return nodelist[nodeid].seq_num_out; /* The packet is data */
      }
}


/* This sends our packet to the synchronization send routine */
int
llc_snd_pkt(nodeid)
int nodeid;
{

   struct ph bph;
   struct mbuf *pkptr;  /* a temp packet header */

   #ifdef FDEBUG
     printf("LLC_SND_PKT:In llc_snd_pkt\n");
   #endif

   /* Get the Pointer to the head of the queue */
   if (nodelist[nodeid].pkt_q == NULLBUF)
      {
        #ifdef MDEBUG
          printf("LLC_SND_PKT:Our Q is EMPTY get out!\n");
        #endif
        return 1; /* our Q is empty */
```

```
      }

   #ifdef MDEBUG
     printf ("LLC_SND_PKT:About to copy llc_pkt_q of size %d and cnt %d\n",
nodelist[nodeid].pkt_q->size, nodelist[nodeid].pkt_q->cnt);
   #endif

   /* allocate some space for the packet */
   dup_p(&pkptr,nodelist[nodeid].pkt_q,0,len_p(nodelist[nodeid].pkt_q));

   #ifdef MDEBUG
     printf ("LLC_SND_PKT:New Packet (pkptr) size %d and cnt %d\n",pkptr-
>size, pkptr->cnt);
     printf("LLC_SND_PKT:Going to remove %d bytes off of pkptr\n",
sizeof(struct ph));
   #endif

   pullup(&pkptr,(char *)&bph, sizeof(struct ph));  /* take our info out */

   #ifdef MDEBUG
     printf ("LLC_SND_PKT:Just pulled off bph now pkptr has size %d and cnt
%d\n", pkptr->size, pkptr->cnt);
   #endif
   wamis_set_header(bph.iface,bph.dest,bph.source,bph.type,pkptr);

   llcq_timer(nodeid,nodelist[nodeid].seq_num_out);

   #ifdef TDEBUG
     printf("LLC_SND_PKT:Ok SIGNAL TIMER\n");
   #endif

   #ifdef FDEBUG
     printf ("LLC_SND_PKT:Leaving llc_snd_pkt \n");
   #endif

   return 0;
}


/* Put the packet in our send Q. */
int
llc_q_pkt(iface,dest,source,type,ptr)
struct iface *iface;    /* Pointer to interface control block */
char *dest;             /* Destination WAMIS address */
char *source;           /* Source WAMIS address */
int16 type;             /* Type field */
struct mbuf *ptr;       /* Pointrt to the packet */
{
    struct ph bph;      /* our header information  */
    int node_id = 0;


   #ifdef FDEBUG
     printf ("LLC_Q_PKT:Got in q_pkt of size %d and cnt %d\n",ptr->size,ptr-
```

```
>cnt);
   #endif

   /* Make sure we aren't trying to send after llc has closed down */
   if (llc_used && (type == IP_TYPE))
    {
    /* First we want to set the parameters, then Q the packet, then send it */

      /* Ok now queue the packet until we get an ack, or it's timeout comes */
      bph.iface= iface;     /* save the iface struct */
      bph.dest = dest;      /* save the destination address */
      bph.source = source;  /* save the source address */
      bph.type = type;      /* save the packet type */
      bph.data = ptr;       /* save the pointer to the packet data */

      #ifdef MDEBUG
        printf ("LLC_Q_PKT:About to add %d bytes onto data which has size %d
and cnt %d\n", sizeof(bph), ptr->size, ptr->cnt);
      #endif

      /* create space in the data portion of the packet for our stuff */
      ptr=pushdown(ptr,sizeof(bph));

      /* save our packet header info in the data part of the packet */
      memcpy (ptr->data,(char *)&bph, sizeof(bph));

      node_id = dest[2];

      enqueue(&nodelist[node_id].pkt_q, ptr);  //enqueue the packet

      #ifdef MDEBUG
        printf ("LLC_Q_PKT:llc_pkt_q mbuf has size %d and cnt %d\n",
nodelist[node_id].pkt_q->size, nodelist[node_id].pkt_q->cnt);
      #endif
      if (nodelist[node_id].pkt_q->anext == NULLBUF)
        llc_snd_pkt(node_id);

      return 0;
    }
  else
    {
      #ifdef FDEBUG
        printf("LLC_Q_PKT: not an IP Packet just send it!\n");
      #endif
        wamis_set_header(iface,dest,source,type,ptr);
    }
  return 0;
}


/* This procedure makes sure we clean up any memory we have acquired */
void
llc_cleanup()
{
```

```
 // If there are any packets in our Q free them.  If someone says stop, then
 // they don't care about the packets in the Q.
 int x;


   for (x=0;x<=numnodes;x++)
    {
      #ifdef PDEBUG
        printf("LLC_CLEANUP:Clean up our Q of Size %d\n", llc_pkt_q->size);
      #endif

      free_q(&nodelist[x].pkt_q);

      #ifdef PDEBUG
        printf("LLC_CLEANUP:Our Q is now length %d\n", llc_pkt_q->size);
      #endif
    }
}

/* This initializes our link level node list structure and the timer Q */
void
init_nodes()
{
 int x;

   #ifdef FDEBUG
     printf("INIT_NODES:  Initializing our node Q's and timers\n");
   #endif

   for (x=0;x<=numnodes; x++)
    {
      nodelist[x].seq_num_in = 1;
      nodelist[x].seq_num_out = 1;
      nodelist[x].pkt_q = NULLBUF;

      timerlist[x].timestamp = -1;
      timerlist[x].timetowait = -1;
      timerlist[x].nodeid = -1;
      timerlist[x].pktnum = -1;

      timerprec[x] = -1;
    }
}


/* Initialize the Link Level Control variables */
int
llc_init()
{
   llc_event = -1;      /* We haven't gotten an event yet */
   llc_trash_pkt = 0;   /* The packet is good by default */
   llc_seq_num_in = 1;  /* the default sequence # is 1 */
   llc_seq_num_out = 1; /* set the seq # for the outgoing packets */
   ack_num = 1;
```

```
   llc_pkt_q = NULLBUF; /* reset the Q */

   LLC_Wakeup_Event=0;   /*1000   start our wait state */
   LLC_Start_Timer=0;    /*1001    Start our llc_timer process */
   LLC_Done=0;     /*1002    Let us know when we are done w/ current packet */
   llc_used = 1;   /* Ok we are ready to start processing events */

   init_nodes();

   return 0;
}


/* This function sets our global var to stop our LLC process. */
int
llcstop(argc, argv, p)
int argc;
char *argv[];
void *p;

{
   llc_used = 0;                    /* Stop our loop we are done */
   llc_event = LLC_Shutdown;        /* we are shutting down! */
   psignal(&LLC_Wakeup_Event, 0);
   alert(pp,1);
   llc_cleanup();
   return 0;
}


void
addtolist(nodenum, packnum)
int nodenum;
int packnum;
{
  int x = 1;
   #ifdef FDEBUG
     printf("ADDTOLIST: adding node %d, and packet #%d to timer list\n",
nodenum, packnum);
   #endif

  timerlist[nodenum].timestamp = msclock();
  timerlist[nodenum].timetowait = Timeout;
  timerlist[nodenum].nodeid = nodenum;
  timerlist[nodenum].pktnum = packnum;

  while (timerprec[x] != -1)
    x++;

   #ifdef FDEBUG
     printf("ADDTOLIST: node #%d is precedence #%d\n",nodenum, x);
   #endif

  timerprec[x] = nodenum;
```

```
}

void
removefromlist(nodenum, packnum)
int nodenum;
int packnum;
{
  int x;
  int y;

   #ifdef FDEBUG
     printf ("REMOVEFROMLIST:removing node#%d from timerlist\n", nodenum);
   #endif

  timerlist[nodenum].timestamp = -1;
  timerlist[nodenum].timetowait = -1;
  timerlist[nodenum].nodeid = -1;
  timerlist[nodenum].pktnum = -1;

  x=1;
  y=1;

  while (timerprec[x] != nodenum)
    x++;

  for (y=x;y<=numnodes - 1;y++)
     timerprec[y] = timerprec[y+1];

  timerprec[numnodes] = -1;

  //check to see if we have any more in the list to timeout. change the
  //timer proc's current working nodetimer and pkttimer.
  checklist();
}


void
checklist()
{
  int x = 1;

   #ifdef FDEBUG
     printf("CHECKLIST: check the timer list out now\n");
   #endif

  if (timerprec[1] == -1)
    {
      #ifdef FDEBUG
        printf("CHECKLIST: Our list is empty\n");
      #endif

      nodetimer = -1;
      return;
    }
```

```
  else
    {
      nodetimer = timerprec[1];
      if (timerlist[timerprec[1]].timestamp + Timeout <= msclock())
     //this packet has also timed out.
   {
      #ifdef TDEBUG
         printf("CHECKLIST:  Packet %d, for node %d has also timed out\n",
timerlist[timerprec[1]].pktnum, timerprec[1]);
      #endif

     /*a fast timeout value so the timer proc can handle this */
     timervalue = 100;
     alert(pp,1);
     while (!sigtimerok)
        pause(100);  /* let the timer proc get out */

     psignal(&LLC_Start_Timer,0);
   }
  else
   {
     timervalue = (timerlist[timerprec[1]].timestamp + Timeout) - msclock();

      #ifdef TDEBUG
         printf("CHECKLIST: Node %d's packet %d still has %d ms left\n",
timerprec[1], timerlist[timerprec[1]].pktnum, timervalue);
      #endif

     alert(pp,1);
     while (!sigtimerok)
        pause(10);

     psignal(&LLC_Start_Timer, 0);
   }
  }
}


/* This function checks to see if we are timing somthing, and if we already
are timing then it queues up the next timer */
void
llcq_timer(int nodenum, int packnum)
{
  int x;  //counter var

   #ifdef FDEBUG
     printf("LLCQ_TIMER: ok Q up node #%d, with pkt #%d\n", nodenum, packnum);
   #endif

  if (sigtimerok == 0)
    {
      //ok we are still timing somthing let's Q up the next one.

    #ifdef TDEBUG
```

```
        printf("LLCQ_TIMER: Ok There is someone already timing so Q me up\n");
     #endif

        addtolist(nodenum, packnum);
     }
   else
     {
       //ok the timer proc is waiting for us to signal it

       #ifdef TDEBUG
         printf("LLCQ_TIMER: Ok no one is timing so go for it! \n");
       #endif

       addtolist(nodenum, packnum);
       nodetimer = nodenum;
       psignal(&LLC_Start_Timer, 0);
     }
}


/* This process is what handles our timeout value for our pakets */
/* this process check to see if our ttl has timed out.  If it does
   it signals llc_send to resend the packet */
void
llctimer(a,b,c)
int a;
void *b;
void *c;
{
   int currnode = 0;
   int currpkt = 0;

   pp = Curproc;

   while (llc_used)
   {
     #ifdef TDEBUG
       printf("LLCTIMER:starting our pwait\n");
     #endif

     sigtimerok = 1;
     pwait(&LLC_Start_Timer);  /* Wait until we get called */
     sigtimerok=0;

     #ifdef TDEBUG
       printf ("LLCTIMER:Start our packet timout timer \n");
     #endif

     if (llc_event == LLC_Shutdown)
        break;


     currpkt = timerlist[nodetimer].pktnum;  /* get the pkt # we are timing */
     currnode = nodetimer; //store who we are timing on
```

```
     #ifdef TDEBUG
       printf("LLCTIMER:Start timer for Node %d's packet %d\n",nodetimer,
currpkt);
     #endif

     if (timervalue >= 1)
       {
         #ifdef TDEBUG
           printf("LLCTIMER:  Going to wait for %d ms\n", timervalue);
         #endif
         pause(timervalue);
       }
     else
       {
         #ifdef TDEBUG
           printf("LLCTIMER: Going to wait for %d ms\n", Timeout);
         #endif
         pause(Timeout);      /* Pause for the timeout value */
       }

     timervalue = 0;

   #ifdef TDEBUG
     printf("LLCTIMER:Nodetimer %d:Currnode=%d, currpkt=%d\n", nodetimer,
currnode, currpkt);
   #endif

   if ((currnode == nodetimer) && (currpkt == timerlist[nodetimer].pktnum))
     {
       #ifdef PDEBUG
        printf("LLCTIMER:Node %d's Packet %d Timed out\n",nodetimer, currpkt);
       #endif

       llc_event = LLC_Timeout; /* Set the Event to Timeout event */
       while (!ready)
         pause(1);
       psignal(&LLC_Wakeup_Event,0); /* signal LLC to wake up */
     }
   }
}


/* This function returns the id of the node we request.  It gets
   the information from the ip address.  128.97.93.3 would be node 3, because
   the last digit is a 3 */
int
get_node_id(int32 addr)
{

  addr = 0x000000FF & addr;

  return addr;
```

```
}

/* Print out Logical Link Control Statistics */
void
llc_stat()
{
   int x;

   for (x=1;x<numnodes+1;x+=2)
   {
     tprintf("Node %d Queue Size: %dNode %d Queue Size: %d\n",x,
len_q(nodelist[x].pkt_q), x+1, len_q(nodelist[x+1].pkt_q));
     tprintf("In Pkt Seq #: %dIn Pkt Seq #: %d\n", nodelist[x].seq_num_in,
nodelist[x+1].seq_num_in);
     tprintf("Out Pkt Seq #: %dOut Pkt Seq #: %d\n\n",
nodelist[x].seq_num_out, nodelist[x+1].seq_num_out);
   }

   for (x=1;x<numnodes+1;x++)
    {
      tprintf("Timerprec[%d] = %d\n", x, timerprec[x]);
    }
}
```