002d9ce8-0

COLLABORATORS

	TITLE : 002d9ce8-0	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		August 24, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

002d9ce8-0

1.1 AMIPX Manual

1.2 Disclaimer

Disclaimer.

AMIPX is freeware; it may be freely distributed for the greater glory of the Amiga, as long as no files are altered.
Freeware in the case of AMIPX, also includes the right to develop software that uses it - just in case you were not sure of that.

AMIPX is not in any way related to Novell, neither in support, nor in source code; this means that any support requests or hate mail involving AMIPX should be addressed to the author, G.J. Peltenburg, and NOT to Novell.

We make no warranties, either expressed or implied, with respect to the software described in this document, its quality, performance, or fitness for any particular purpose. Any risk concerning it's quality or performance is solely the user's. Should the program prove defective, the user assumes the entire cost of all necessary servicing, repair, or correction and any incidental or consequential damages. In no event will we be liable for direct, indirect or consequential damages resulting from any defect in the software.

1.3 Introduction

Introduction.

AMIPX is designed to make porting PC games to the Amiga a little easier, and thus make it a bit more interesting to publish games for the Amiga again.

To achieve this, AMIPX uses a Programming Interface that is very similar to the PC's IPX API, and uses the same structures to exchange information between the application and the library. Some differences remain, though.

1.4 When to use it

When to use AMIPX.

AMIPX is best used for porting PC IPX networking code to the Amiga. It is NOT the protocol of choice for new games. You should use TCP/IP for those, since that makes it possible to use the Internet as well as a LAN for playing head-to-head.

AMIPX does not currently support IPX checksumming, SPX, SAP, or Routers, and is therefore of little use for serious software (such as Netware clients).

1.5 Past

Late november, 1997:
I decided to write IPX support for the Amiga, after failing to find existing software say from Novell.

Sending and receiving IPX packets is quite simple on an Amiga, all it requires is a Sana-II device driver, and some knowledge of the way IPX data is organised. I wrote a program that monitored and sent IPX packets, in less than two weeks, starting from scratch, and without prior experience with Sana-II.

Making the API almost the same as that of the PC, and putting all that into a shared library is a lot more work, and it took me some 6 weeks to complete it.

18th of january,1998: First Aminet release. Actually took another week due to Aminet trouble.

29th of january,1998:
  Started work on version 1.1. Purpose: to remove AMIPX' dependency on the packet filtering function, that seems to be lacking in many Sana-II drivers, and to make it easier to make future modifications such as multiple network adapters and internal bridging by AMIPX.

1st of february,1998:
  Version 1.1 now sends and receives packets.

2nd of february,1998:
  SendTask failed to free its IORequest. Fixed.
  Tested with liana.device. Works in this version of amipx.

1.6 Present

What AMIPX currently does:

 1) It sends and receives IPX packets and distributes the received packets
    to the correct application.

 2) It offers almost the same basic API as PC IPX.

What it does not do:

 1) Use routers to access other network segments.

 2) Support SPX.

 3) IPX checksumming.

 4) Support other frame types than Ethernet.

1.7 Requirements

  AMIPX requires AmigaOS3.0. It may run on 2.04, but that has not been tested.

  In addition, it requires a Sana-II network driver. It no longer needs, nor uses packet filtering.

  Also, AMIPX currently only supports Ethernet frame types, and therefore it does NOT yet work with Arcnet hardware, or point-to-point Amiga to PC networks.

  amipx.library will not work without a valid amipx.prefs file in ENV: that file can be created with the prefs editor included in this distribution.

  AMIPX should run on all current 68k processors. No PPC version is planned yet.

```
AMIPX coexists with...
----------------------
AmiTCP/IP - even on the same network adapter. Should be the case with Envoy and
            any TCP/IP stack, since the packet types used by IPX differ from
            those of TCP/IP.


Tested Sana-II device drivers...
--------------------------------
Ariadne.device v1.44  - works fine (my own Ethernet card)
Liana.device   v1.4   - works but performance depends on processor. Set frame
                        type to Ethernet-II and choose two different node
                        addresses.

Yeah, just two - I'd like to extend the list, but I need your comments for
that. I/m especially interested in Ethernet driver compatibility - after all,
AMIPX was designed for Amiga to PC networking.
```

1.8 Installing AMIPX

```
Told you it was easy:

Step 1: copy amipx.library to your LIBS: drawer.

Step 2: copy amipx_prefs to your Preferences drawer.

Step 3: use amipx_prefs to set up AMIPX for your network.

You may wish to install AMIPX in more than one bootable partition on your
hard drive. If you do, be sure to copy the file amipx.prefs file from ENV:
as well, or to set up AMIPX using the preferences editor after booting from
that other partition.
```

1.9 Preferences

```
Basics
------
AMIPX requires you to set it up with specific information on how to access
the network card, and information on your network, before it can do its
stuff.


Fields
------

Network number:
Should be set to the IPX network number of your LAN - which is usually 0
in a LAN consisting of a single segment.


Node:
```

Used only for network drivers that do not have a default network address.
It must be unique within the segment you are connected to.
Ethernet network cards have their own default addresses, so you will not need
to set this up for those.

Device:
The location and name of the Sana-II network driver to use. Use the '?' button
to pick the driver.

Unit:
The unit number of the network interface to be used with the Sana-II driver -
0 if only one network interface of its type is installed in your computer.

Frame Type:
This determines the makeup of the data part of the packets, and must be set
to the same type as the other computers are using.

Saving or Using the modified preferences

Since AMIPX is a shared library, it is set up when the first application
opens it. At that time, the Sana-II device driver is opened, and the
preferences are read. When the last application closes the library, the
device is closed.
Because of this, depending on the applications running, you may need to
quit them or tell them not to use AMIPX, so that they close the library,
before the new settings take effect. If an IPX application has crashed
(and therefore has not closed the library), rebooting the Amiga is necessary.

1.10 Future

I intend to add the ability to use routers, so you can play over a WAN as well.

I also intend to add support for AMIPX to use more than one network interface.

1.11 Credits

Thanks to:

Máximo Piva for his comments

ClickBoom for their positive response

The late Commodore-Amiga, Inc., for their near-perfect Sana-II standard.

AMIPX was written with Manx Aztec C 5.0, and incorporates its resident
library startup code - not a single assembly instruction was needed.

```
How to Contact the author
-------------------------
Email: gpeltenburg@wxs.nl
```

1.12 AMIPX Applications Programming Interface

```
AMIPX API


TABLE OF CONTENTS

AMIPX_GetLocalAddress
AMIPX_OpenSocket
AMIPX_CloseSocket
AMIPX_SendPacket
AMIPX_ListenForPacket
AMIPX_RelinquishControl
AMIPX_GetLocalTarget

ECB description
IPX header description
Additional requirements


    NAME
  AMIPX_GetLocalAddress -- obtain own network address from IPX driver.

    SYNOPSIS
        AMIPX_GetLocalAddress(addressspace);
            a0

        VOID AMIPX_GetLocalAddress(BYTE addressspace[10]);

    FUNCTION
        This function copies the full internetwork address into the supplied
        array, in transmission order.

    INPUTS
        addressspace  - address buffer of at least 10 bytes.

    RESULTS
        none - it is a simple copy operation

    EXAMPLE

    NOTES




    NAME
```

```
AMIPX_OpenSocket -- open an IPX socket.

  SYNOPSIS
      actualsocket = AMIPX_OpenSocket(requestedsocket);
d0        d0

      WORD AMIPX_OpenSocket(WORD);

  FUNCTION
      This attempts to open a socket with the requested socket number, or
      assigns a free socket number if requestedsocket == 0.

  INPUTS
      requestedsocket - socket number

  RESULTS
      actualsocket will be 0 if the socket table is full, or if the requested
      socket is already in use.

  EXAMPLE
      mysock=AMIPX_OpenSocket(0x869c); // DOOM socket number
      if(mysock==0)
       panic();                        // socket already in use



  NAME
AMIPX_CloseSocket -- close an IPX socket.

  SYNOPSIS
      AMIPX_CloseSocket(socketnumber);
      d0

VOID AMIPX_CloseSocket(WORD);

  FUNCTION
This closes the socket and cancels any outstanding read requests.

  INPUTS

  RESULTS
none - socket will be closed if open

  EXAMPLE
AMIPX_CloseSocket(mysocknum);

  NOTES
To cancel outstanding Listen ECBs (read requests), you should close
      the socket with this function. You must NOT attempt to manually change
the ECB's status and subsequently call AMIPX_ListenForPacket in order
to cancel a read request. If you were to do so, on receipt of a packet
for the socket, innocent memory may be overwritten.



  NAME
```

```
AMIPX_SendPacket -- send an IPX packet.

  SYNOPSIS
error = AMIPX_SendPacket(ECB);
d0       a0

WORD AMIPX_SendPacket(struct AMIPX_ECB *);

  FUNCTION
This submits a send request to the network card. The call returns
      immediately; by checking the ECB's InUse flag (0 means ready), the
IPX user can see if the packet has been transmitted yet.

  INPITS
ECB - pointer to Event Control Block.

  RESULTS
error - 0 if all went well.

  NOTES
Currently, the only reason why AMIPX_SendPacket might fail is lack of
memory.




  NAME
AMIPX_ListenForPacket -- Insert ECB in IPX read queue/remove it.

  SYNOPSIS
error = AMIPX_ListenForPacket(ECB);
d0            a0

WORD AMIPX_ListenForPacket(struct AMIPX_ECB *);

  FUNCTION
      Inserts ECB in listen queue.
You should initialise the InUse flag of the ECB to 0x1d, then call
      this function.
The InUse flag can be checked to see if a packet has been received.
      InUse will be 0 once a packet has been received.


  INPUTS
ECB - pointer to Event Control Block

  RESULTS
error - 0 if successful.

  EXAMPLES

... // initialise ECB
myECB.InUse=0x1d;
if(AMIPX_ListenForPacket(&myECB)) // insert in queue
 panic(); // out of memory!
```

```
...
if(myECB.InUse==0) {
 ...      // process packet
       AMIPX_ListenForPacket(&myECB);   // insert again
      }
```

NOTES
You must NEVER change the InUse flag of an ECB when it is in the queue.
This call will fail if out of memory, or if the socket has not been
opened.

NAME
AMIPX_RelinquishControl -- allow for some IPX housekeeping

SYNOPSIS

```
AMIPX_RelinquishControl();


VOID AMIPX_RelinquishControl(VOID);
```

FUNCTION
This will allow the IPX library to clean up processed send requests.

INPUTS

RESULTS

EXAMPLES

```
AMIPX_RelinquishControl();
```

NOTES
This cleanup is also performed when any other AMIPX function is
called.

NAME
AMIPX_GetLocalTarget -- find network address to send a packet to

SYNOPSIS

```
error = AMIPX_GetLocalTarget(internetworkaddress,localtarget);
d0           a0      a1

      WORD AMIPX_GetLocalTarget(BYTE internetwaddr[12],BYTE localtarg[6]);
```

FUNCTION
This function attempts to find the node number on the local network,
to which a packet has to be sent, in order to have it end up
at the requested internetworkaddress. This can be used to find the
router/bridge on the local network, that leads to the target network.
The ImmedAddr field of the ECB can then be filled with the local target.

INPUTS

```
internetworkaddress - an array of 12 bytes, consisting of network
          number (4 bytes), node number (6 bytes) and
```

```
            socket number (2 bytes). This is the end address
            where you would want your packet to arrive.

    RESULTS
  error - 0 if successful, nonzero if not found for some reason.

  localtarget     - an array of 6 bytes, filled in with the node
            number of the device on the local network to
            which you should send the packet.

    EXAMPLES

    NOTES
  Currently, not really operational, but succeeds if target network
        number is equal to that of the amiga, and fails if not.
```

ECB - Event Control Block

The ECB contains all the information on a Send or Receive request, that is needed by the IPX library, and it is the ECB, that is the real interface.

```
    FIELDS
  ESR       - Event Service Routine, see below. Make NULL unless you
                    provide an ESR.
  InUse     - will be zero once the ECB has been processed.
  CompletionCode - nonzero if an error occurred while processing.
  Socket    - the number of the socket the request should be queued in.
  IPXWork   - reserved for internal use, subject to change and sensitive.
  DWork     - ditto
  ImmedAddr - Node address to send the packet to, ffffffffffff for
        broadcast.
  FragCount - Number of memory fragments that follow.
  Fragment[n] - record:
      FragData - pointer to buffer
      FragSize - size of buffer

  Note that in a Send, all fragments are concatenated to form the actual
  IPX packet. In this case, the size of the transmitted packet is equal
  to the sum of the fragment sizes.

  In a Listen, the fragments are filled starting at the first fragment,
  and if that is full, the second, and so on. In this case, the total
  packet size is only available in the IPX header, since FragCount and
  the fragment definitions are not modified by ListenForPacket.

  Also note that the ECB is an internal structure, and you must therefore
  simply assign all fields without resorting to 'swap' functions.

    Event Service Routine

  The ESR is an optional pointer to a procedure (i.e. no return value)
  that is called when the ecb has been processed.

        void (*ESR)(BYTE caller,struct AMIPX_ECB *ecb)
                         D0                      A0
```

The procedure when called, will have two arguments:
caller - indicates which software calls it - currently only IPX
is supported, indicated by a value of 0xff.
ecb - the address of the ecb. The s

Why use multiple fragments?

With multiple fragments, sending and receiving packets will be only a fraction slower.
The advantage of multiple fragments is: you can split the header from your own data, by assigning a pointer to a header as fragment 0, and assigning a pointer to your 'user buffer' to fragment 1. You can then access the header and the user data separately. Moreover, you might make a table of initialised headers, and send the same data to all of those targets, without modifying a lot of data - all you need to do is set up the ECBs.

Initialisation

The following fields in the ECB must be initialised before calling AMIPX_ListenForPacket:

ESR must be set to NULL, or to the address of a function to be called once the packet has been processed.

InUse Should be set to 0x1d on the first call to AMIPX_ListenForPacket. On subsequent calls, this field is kept up to date. However, if you use the same ECB for both listening and sending, you MUST set it to 0x1d again after the send has been processed.

FragCount must be set to the number of memory fragments.

Fragment[n] must contain a pointer to a buffer, and the size of that buffer.

The following fields in the ECB must be initialised before calling AMIPX_SendPacket:

ESR must be set to NULL, or to the address of a function to be called once the packet has been processed.

ImmedAddr must be set to the node address to which the packet is to be transmitted. In a single segment network, this is the node-part of the destination address.

FragCount must be set to the number of memory fragments.

Fragment[n] must contain a pointer to a buffer, and the size of that buffer.

IPX header

The IPX header, which consists of 30 bytes, is filled by the IPX user, and contains information such as destination address and source address.

FIELDS

Checksum can contain a checksum for corrupt packet detection, but does not seem to be used by most software. This should be set to 0xff,0xff.

Length holds the total length of the packet, including the 30 bytes of the header. First byte holds the most significant bits, second holds the least significant bits.

Tc may be filled in by IPX - apparently holds the number of hops (transfers from one network segment to another) it took to get here.

Type This is the IPX packet type - don't confuse with frame type. The packet type should be set to 4 (packet exchange) although PC DOOM sets it to 0 (unknown packet).

Dst This is the complete internetwork address (network,node, socket) of the destination.

network	network number, MSB first, 4 bytes.
node	network card number, MSB first, 6 bytes.
socket	socket number, MSB first, 2 bytes.

Src This is the complete internetwork address (network,node, socket) of the sender.

Note that all fields in the IPX header should be defined as bytes or byte arrays, or that system dependent 'swap' functions should be used to assign or read these values. Although IPX uses the same byte ordering as the amiga, source code would not be portable to other architectures if you didn't.

Also note that, although not required, most games use the same socket for the source as the target. The reason is of course, that broadcasting separately to obtain the socket number to send to, then sending to that socket, is a lot more unnecessary work.

Initialisation

Before AMIPX_ListenForPacket, NO FIELDS need to be initialised in the IPX header.

Before AMIPX_SendPacket, all fields should be initialised (see above), Tc should be set to 0.

Additional requirements

OpenLibrary only by processes

The first task calling OpenLibrary for amipx.library, MUST be a DOS or CLI process. This is because the first opener, also reads the communications

preferences. Additional tasks may call OpenLibrary later, and as long as the library is open, it will work fine.

First opener and last 'closer' must have free signalbit
The initial communications between the library and the subtasks is done through Exec Signals. One is needed at opening time and at closing time. Again, if your task is NOT the first to open the library, and it is NOT the last to close the library, this is of no concern. Especially beware of not freeing a signal if you aren't sure if your program will be called repeatedly from one CLI.