ModemLink

Michael Veroukis

COLLABORATORS			
	TITLE : ModemLink		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Michael Veroukis	July 26, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

ModemLink

1.1 ModemLink device - Table Of Contents

1.2 ModemLink device - Introduction

Introduction:

~~~~~~~~~~~~

The ModemLink device was written in an attempt to make modem linkable games easier to write on the Amiga. We've seen hundreds of games on the PC with modem play capabilities, but most Amiga games don't provide this option. This frustrates many Amiga game players, as everyone knows it's much more fun to kill your friends then stupid computer-AI people!

It's no secret that most Amiga programmers are not part of some rich company. Therefore, it's easy to understand why many developers don't add
~~~~~~~~~~~~

modem play to their games.

First of all, you may be adding a few months to development time just for the modem link part of the game. For many hobby programmers, they'd rather just finish the game quickly and move onto new and exciting things. The ModemLink device gives the developer all the code required for a basic modem playable game, therefore saving on development time.

Another problem is that you'd need two Amigas, with a modem on each, and preferably two phone lines (although you could always use a null modem as well). Not everyone can have this kind of set up. The ModemLink device makes it a little easier to develope the modem linkable game, since there's less to worry about. You'll still need to test, but it's the sort of thing a friend with an Amiga and a modem can help out with. Since the ModemLink device has been tested, and is known to work, a game developer only needs to worry about how to interface with the ModemLink device.

Well, this is all very cool and all, but how does it work??? Although I don't want to get into the finer details of the device, this is a good place to give a rough description of the thing.

The ModemLink device provides two important functions; Modem Connectivity and a Reliable Serial Protocol. What does this mean???

Modem Connectivity:

~~~~~~~~~~~~~~~~~~

The ModemLink device provides standard routines to dial/answer calls. These routines can be used to easily set up a modem connection. They have been designed to be very flexible and provide significant control to the programmer.

Reliable Serial Protocol:

~~~~~~~~~~~~~~~~~~~~~~~~

Once a modem connection is made, data must be sent and received across the phone line. One needs to be sure that the data received is correct and the data sent has been received. To do this, some kind of protocol must be used, which the ModemLink device provides. Although designed to support more protocols in the future, it currently only supports a single protocol based on what is called the Stop And Wait protocol.

Anyway, as you might have guessed, the main point to this entire project is to make it easier for Amiga programmers to make modem linkable games. I offer this package (with source included) free for everyone to use.

1.3 ModemLink device - Copyright and Distribution notice

Copyright:

~~~~~~~~~

1.4 ModemLink device - Requirements

Requirements:

~~~~~~~~~~~~

The ModemLink device requires the following:
1) Amiga Computer
2) AmigaOS 2.04 or higher
4) C compiler (works very nicely with SAS/C)
Another Amiga and a modem for each is also highly recommended.
I found out that StormC has some problems compiling the source. I don't
have the registered version of StormC so I'm not currently able to provide
StormC source. If anyone is up to the challenge of doing a port, let me know.
A StormC pragma file is however provided in the include/StormC_pragmas
directory.

1.5 ModemLink device - Contents

Contents:

~~~~~~~~

In this archive you will find:
ModemLink/
~~~~~~~~

AutoDocs/
ModemLink.doc
Devs/
modemlink_000.device
modemlink_020.device
Docs/
ModemLink.guide
Examples/
Debug
DeviceStuff.c
DeviceStuff.h
SCoptions
smakefile
TestML
TestML.c
TestMLDev
TestMLDev.c
include/
clib/
ModemLink_protos.h
ModemLink/
ModemLink.h
pragmas/
ModemLinkDev_pragmas.h
proto/
ModemLink.h
lib/
ModemLink_000.lib
ModemLink_020.lib
ModemLinkDev.lib
Source/
CRC.c
CRC.h
Debug
dev/
DeviceStuff.c
DeviceStuff.h
lib/
Link.c

Link.h

LinkDevTags.c

LinkDevTags.h

LinkTags.c

LinkTags.h

Modem.c

Modem.h

ModemDevTags.c

ModemDevTags.h

ModemLinkAPI.c

ModemLinkAPI.h

ModemLinkDev.c

ModemLinkDev.fd

ModemLinkDevAPI.c

ModemLinkDevAPI.h

ModemLinkDev_pragmas.h

ModemLinkTask.c

ModemLinkTask.h

ModemTags.c

ModemTags.h

SCoptions

smakefile

1.6 ModemLink device - Installation

Installation:

~~~~~~~~~~~~

There is no installer that comes with ModemLink device, but that's okay,
it's not hard to do by hand. So, here's what you should do to install this
thing:

1) Use LHA to unarchive the ModemLink archive (which should have been done
by this point. A new directory will be made by LHA called ModemLink.

2) Copy the new ModemLink directory anywhere you like. It might make the
best sense to put it somewhere where you keep development stuff...
Somewhere near your C compiler perhaps.

3) Copy one of ModemLink/Devs/modemlink_XXX.device (where XXX indicates the
CPU your using) into your DEVS: directory as "modemlink.device".

4) Copy one of the ModemLink/lib/ModemLink_XXX.lib to where ever you like
to keep your link libraries. Of course, you don't need to perform this
~~~~~~~~~~~~

step if you don't want to. You can instead just give the full path name
to your linker in the make files.
5) Copy the directories/files in ModemLink/include to where ever you keep
your C include files (include:). This too is optional, as it is not
necessary. Alternatively you could use the IDir command line option
with SAS/C or just specify the full path when including the files in
the source code.
That's really all there is to it. Of course, you can scatter the rest of
the archive around, however, once you start doing that, compiling the source
and example files may require some smakefile modifications.

1.7 ModemLink device - Technical Information

Technical Information:
~~~~~~~~~~~~~~~~~~~~~~
Here I will attempt to describe the basic design behind ModemLink, along
with an explanation on how to use it. Since there are many different issues
to discuss, I've decided to break it down into sections.
If you need an in-depth description of each command/function the ModemLink
device provides, please see the ModemLink autodoc. For in-depth examples
please look at Examples/TestML.c and Examples/TestMLDev.c as well as the
Source/ directory.
Basic Design - A description of how everything fits
together at a high level.
Device vs Link Library - Although initially intended to be a device
I've also made a link library with the same API.
The main reason this exists is that some
people prefer link libraries over .libraries
or .devices. Now you have a choice!
Where To Begin - Describes how to initialize the ModemLink
device.
Modem - This describes the routines used to dial or
answer calls.
Link - A description of how to set up a protocol
and how to use it to send data packets back
and forth.
Examples - In this archive I've included two small example
programs. This section quickly describes them.
Source - Complete source has been included with this
archive.
~~~~~~~~~~~~~~~~~~~~~~

1.8 ModemLink device - Tech Info/Basic Design

Basic Design:

~~~~~~~~~~~~

As it's name implies, the ModemLink device is an Amiga device, not just
a link or runtime library. I decided that it made the most sense to stick
the ModemLink protocol in a device because the device API lent itself very
well to what I needed.
First of all, devices allow for asynchronous IO very nicely. This was my
main concern. Libraries aren't really designed for Async-IO. To do so,
I'd pretty much have to re-create the same programming method you'd use for
a device. So what would be the point???
Another reason is that the device API is very nice and clean and most
people are very familiar with it. I realized that there would have to be
a set of routines that a programmer could call directly, so the device
also has functions (like the timer.device).
When I first set out, I wanted to make a very basic protocol, and
nothing else. I later realized that some support routines would be really
nice too. Therefore, I added the modem support routines.
When making the modemlink protocol, I decided to use a system which sends
and receives packets of data. This way, a large block of data can be sent
across the link as a set of smaller packets.
Sending a large block in smaller packets has an important advantage;
If an error occurs, the protocol will re-send only the packet that was
in error. Errors could result from line noise.
More support functions were created to help with the handling of the
packets. These can take a large data block and split it into a linked
list of packets or vica verca.

1.9 ModemLink device - Tech Info/Device vs Link Library

Device vs Link Library:

~~~~~~~~~~~~~~~~~~~~~~

Although the ModemLink device was designed to work like an Amiga runtime
device, I decided to make it into a link library as well.
Why would I want to do this??? Well, first of all, it was a lot easier
to test it while developing. If I developed it as a device from scratch,
I'd have to copy my test program AND the device to my test machine.
So, it was much more convenient to just have the one file with all the

routines built into it. I felt it also made debugging easier.
However, another reason to support link libraries is that many people don't like sticking yet another xyz.device file into their DEVS: directory. And when you consider that not many people will be playing too many modem linkable games at the same time, the argument for code reuse kinda falls apart.
However, the advantage of a device that links with your program at run time is that if I make any bug fixes, the programs that use ModemLink device do NOT need to be re-compiled.
To maintain the same API between the device and link library I had to re-create some of exec's basic IO request handling routines. Since some of exec's routines expect to be dealing with a real device, they simply won't work when using the link library. The exec.library functions that should NEVER be called when dealing with the ModemLink.lib are:
a) AbortIO()
b) DoIO()
c) SendIO()
Instead, use the following routines to achieve the same results when using the link library:
A) ML_AbortIO()
B) ML_DoIO()
C) ML_SendIO()
These replacement routines function in the same way as their corresponding exec.library ones. The only difference is that they by-pass the normal device interface and call the routines that do all the work directly. Make sure to ONLY use these replacement routines when using the link library and never with the modemlink.device.
So, my advice is to think vary carefully what you wish to use. I'd recommend the device over the link library, but it's up to you.
NOTE: There is also the ModemLinkDev.lib link library which contains the glue routines for all the ML_xxxTags() calls in the modemlink.device. If you wish to use routines like ML_DialModemTags() (instead of ML_DialModemTagList()) then you must make sure to link with the ModemLinkDev.lib. This is not necessary if you use the ModemLink.lib as it already contains them.

1.10 ModemLink device - Tech Info/How to get started

Getting started:

~~~~~~~~~~~~~~~~

Before you can use the ModemLink device you must first use exec.library/ OpenDevice() to open and initialize the ModemLink device. Since the ModemLink device also has procedures that a program can call directly, make sure to get the base address of the device. Also, remember to make a call to exec.library/CloseDevice() once you're done with the ModemLink device. Currently the ModemLink device doesn't support sharing of a device unit. However, to get around this you can make a copy of the IOExtIO structure initialized by OpenDevice(). For more details see the RKM Devices, as well as the Link Section.

Of course, if you're using the link library you don't need to do any of this. However, you'll still need to create a message port and an IORequest (struct IOExtLink).

For more details please take a look at example one and example two and the RKM Devices book.

1.11 ModemLink device - Tech Info/Modem

Modem Section:

~~~~~~~~~~~~~

The modem section requires that the serial.device (or some similar device) has already been opened (via exec.library/OpenDevice). All the modem routines described here will need to use the IOExtSer structure initialized by OpenDevice.

The section of the ModemLink device that deals with direct modem support consists of three basic routines;

a) ML_AnswerModemTagList(SerIO, tags);

b) ML_DialModemTagList(SerIO, PhoneNum, tags);

c) ML_SendModemCMDTagList(SerIO, CMD, tags);

The first two (a & b) are specialized routines designed to only do what their name implies. The third one (c) is designed to send any command to the modem and wait for a result code.

As a safety feature, all modem related routines will timeout after a set amount of time. The amount of time depends on the routine. It is however possible to set the timeout value using the ML_DialTime or ML_AnswerTime tags.

All three routines will return one of the following return values:
1) MODEM_OK - Modem returned "OK". Means command was successful.
2) MODEM_ERROR - Modem returned "ERROR". Means there was an error
with the command issued. Most likely the command
issued was misspelled or just down right wrong!
3) MODEM_BUSY - Modem returned "BUSY". Obviously, the number you're
trying to dial is in use, and therefore BUSY! This
is usually accompanied by the most horrible sound
coming through your modem speaker!!! (is there
anything worse then the busy signal???)
4) MODEM_NOCARRIER - Modem returned "NO CARRIER". Something bad
happened with the last command. Maybe try again.
5) MODEM_NODIAL - Modem returned "NO DIALTONE". This is pretty bad.
Indicates the modem can't get a dialtone when it
goes Off-Hook. Make sure modem is connected to a
line!
6) MODEM_OFF - Can't detect modem. This is mostly caused by modems
which are not powered up or are not connected to
the computer. Fix this!
7) MODEM_CONNECT - Carrier Detected! Happy! Happy! This means the
modem has actually connected with another modem over
the phone line. Now the fun begins!
8) MODEM_TIMEOUT - Nothing happened! Timeout has been triggered by the
timer device. This means the modem command issued
could not finish in the amount of time provided.
You may want to try again, or set a higher timeout
value.
Make sure to always check the result code and be prepared to deal with
all possibilities. It is always a good idea to provide some kind of user
feedback when dialing/answering so relay the results to the user.
To answer an incoming call ML_AnswerModemTagList() is used. It will issue
the auto-answer command to the modem and then wait for carrier detect or
time-out. The default auto-answer command used is "ATS0=1". Other commands
can be used with the ML_AutoAnsText tag. Before issuing the auto-answer
command to the modem it will append the Suffix string (default: "\r"). To
change the Suffix appended to the auto-answer command use the ML_Suffix tag.
To place an outgoing call, ML_DialModemTagList() is used. It will send the
modem a dial command including the phone number passed to
ML_DialModemTagList(). The dial command will be prefixed with "ATDT " and "\r"

is added at the end. The prefix can be changed with the ML_DialPrefix tag, while the suffix can be changed with ML_Suffix tag. Once the dial command is sent to the modem it will wait for carrier detection. If it does not detect the carrier within the timeout interval (can be specified via the ML_DialTime tag) it will abort the dial and return MODEM_TIMEOUT.
Both the dial and answer routines use ML_SendModemCMDTagList(). This routine does all the real work. It will take any modem command, issue it to the modem and wait for some kind of result. This can be used to send the initialization string to the modem, or to hang up the modem. It can also be used to dial and answer incoming calls (if you don't like the built in dial/answer routines).
For more details on how these routines work and a complete description on the tags each use please see the modemlink autodoc file.
Once you have connected with the modem, you can then establish a protocol and begin communicating over the link. However, you do not need to use ModemLink's built in routines to set up a modem connection. The system is designed so that custom dialing/answering routines can be used.

1.12 ModemLink device - Tech Info/Link

Link Section:
~~~~~~~~~~~~
This section of the ModemLink device deals with data transfer. This includes packets and the ModemLink protocol. It is important that you be familiar with the Amiga's standard device API as the ModemLink device uses it (see the RKM Devices book for an in-depth explanation of the device API).
Once you have a modem connection (or null-modem hookup), you then have to establish a protocol. The routines dealing with starting and stopping a ModemLink protocol are:
a) ML_EstablishTagList(LinkIO, SerIO, tags);
b) ML_Terminate(LinkIO);
ML_EstablishTagList() will launch the handler task that will maintain the ModemLink protocol. Since it communicates directly with the serial device, it will make a copy of SerIO for it's own use. It will also initialize the LinkIO structure. LinkIO is the IORequest struct that you can now use to send IO requests to the device. If you want more then one LinkIO struct follow these steps (and check out this example):
- You will need another MsgPort (MP2) and another LinkIO struct (LinkIO2)
1. Create the new MsgPort (MP2)
~~~~~~~~~~~~

2. Copy over the initialized LinkIO struct over the new one (LinkIO2)
3. Set the ReplyMsg port in LinkIO2 to point to MP2.

You can now use both LinkIO structs to send and receive data from the device. It might be a good idea to have one LinkIO for reading and another for writing.

To end the protocol link, make a call to ML_Terminate(). Make sure to send it an initialized LinkIO.

Once the modem link protocol has been set up, IO requests may be sent to the device for processing. For a full description of each command please see the autodoc file. Here I will only go over some important features that may need some clarification.

The ModemLink device sends packets of data back and forth. The packet structure it uses is defined as follows:

```
struct LinkPkt {
struct MinNode ml_Node; // for linked lists
ULONG Length; // size of Data block
ULONG CRC; // contains CRC32 code (internal use)
UBYTE Socket; // not used - set to zero
UBYTE *Data; // points to data block
int Flags; // no flags yet - set to zero
UBYTE *UserData; // points to user defined data
};
```

There are 4 IO commands that are used to send and receive data from the device. These are:

a) CMD_READ
b) CMD_WRITE
c) MLCMD_READ
d) MLCMD_WRITE

The primary difference is that the first two (a & b) deal with the raw data only (they create a LinkPkt structure internally), while the last two (c & d) deal with LinkPkt structures exclusively. This means you have a choice in how to communicate with the device. The first two commands (a & b) operate similar to how other devices do (ex. serial.device). However, since this adds a little overhead, you may want to optimize it by re-using a previously allocated LinkPkt for each request. But more importantly, using LinkPkt's allows you to use the packet handling functions which make dealing with large read/write operations easier. This is why the last two (c & d) commands were created.

Another important thing to understand is the difference between the

read (a & c) and write (b & d) commands. The read commands will send an IO request with a NULLed out io_Data field to the ModemLink handler task. Once a packet comes in, the handler task will allocate memory for the data, stick it into the read IO request, and ReplyMsg() it back to the user program. This way, every time a read request is satisfied a new LinkPkt structure is allocated (or in the case of the CMD_READ just the data block). Therefore, it is up to the user program to deallocate the packet and/or data block once it is no longer needed.

However, when using the write commands the io_Data block will point to either an initialized LinkPkt or just to the raw data block. Once the handler task has sent the data it will ReplyMsg() the IO request back to the user program. Once this is done it is safe to deallocate or re-use whatever io_Data points to.

To make living with packets easier the ModemLink device includes several built in routines that handle LinkPkts. These are:

a) ML_AllocPkt();

b) ML_FreePkt(Pkt);

c) ML_FreePktList(PktList);

d) ML_PacketizeData(PktList, Data, Length, PktSize);

e) ML_DePacketizeData(PktList, Data, Length);

f) ML_PacketDataSize(PktList);

The first three (a, b & c) deal with packet allocation and deallocation. It is recommended (although not necessary) to use these routines whenever allocating or deallocating LinkPkts. These are fairly straight forward. For more information on these please see the autodoc file.

Although routines d & e also allocate and deallocate LinkPkts they are a bit more specialized in their use. ML_PacketizeData() will take a large chunk of data, and split it into a series of packets. Each new packet created will contain a copy of a segment of the original data chunk. The maximum data block for each new packet created is specified by the PktSize parameter.

The ML_DePacketizeData on the other hand will take all the data from a list of packets and copy them into a larger memory chunk. The memory chunk must be already allocated. To calculate the size of the new memory chunk use the ML_PacketDataSize() routine which will traverse a linked list of packets and return the total amount of data.

These routines can be very handy when you need to send a large block of memory over the ModemLink device. Remember, if an error occurs while sending a packet, only the packet with the error will be resent by the

protocol. Therefore, smaller packets will re-send faster. However, the more packets there are, the more overhead is required to send. So, a balance must be found. A good rule of thumb is to set the maximum size of your packets to about the same as your baud rate. So, for a 14.4 modem a maximum packet size of about 1400 bytes would be fine. Feel free to experiment.
As you can see, the Link section is somewhat complicated. However, it does ensure that each packet will arrive to it's destination without error.
It allows for Async-IO or Sync-IO through the Amiga's standard device API.
Some work is still required to make it better, but the basic parts are here.
NOTE: Currently the ModemLink device supports only one protocol; Stop and Wait. This means it sends a packet, then waits for acknowledgement from the other side before it sends another packet. This is not ideal for high through-put applications. However, the ModemLink device was designed for future expandibility. The ML_EstablishTagList() routine may one day take tags to allow for different protocols (such as the sliding window protocol which is much nicer for high through-put applications).

1.13 ModemLink device - Tech Info/Examples

Examples:
~~~~~~~~
With this archive two example programs were included (TestML.c & TestMLDev.c). Currently, the only difference between the two is that TestML was created using the ModemLink.lib and TestMLDev uses the modemlink.device. I highly recommend you take a close look at these.
I also recommend to play around with them and try out different commands (after all, these are the exact same test programs I used to test every feature)
The programs both function in the same way. They are a very crude attempt at a terminal program using the ModemLink device. To test them out, you'll need two Amigas, with a modem and phone line for each. Make one dial the other.
USAGE: TestML <phone_number>
This will make TestML dial the number given. If the number is busy it will try two more times to get through before it gives up. If no phone number is given it will instead set auto-answer on and wait for a call.
Be warned that it will timeout eventually if it doesn't connect.
Once the modem has connected the program will immediately establish a
~~~~~~~~

protocol. Once this is done, it will give you a prompt. You may now type in a message. Hit return to send it across the modem link. Please note that any messages that come in will not be displayed until you hit return. Therefore, you must keep hitting return every now and then to check for a new incoming message (Yeah, I know this sucks, but it works great for testing :).
To quit the program just type a period and return on a new line. This will send a kill signal to the remote site and shut it down too. And that's it!

1.14 ModemLink device - Tech Info/Source Files

Source Files:
~~~~~~~~~~~~
As I've already mentioned, the complete source to the ModemLink device can be found in the Source/ directory. A smakefile is provided for easy re-compilation. Note that if you make any changes and re-compile it that you'll have to move the final #?.device and #?.lib to where you keep the current ones.
If you look into the Source/ directory, you'll notice that some of the files are also in the include/ directory. In fact, the include/ModemLink/ ModemLink.h file is actually a combination of some of the include files in the Source/ directory. So, if you make any changes to the include files in one of the two, make sure to reflect the changes to the other copies before re-compiling. Also, if you're looking for the .fd file for the device, it's kept in the Source/ directory.
Happy Hacking!

1.15 ModemLink device - Future Enhancements

Future Enhancements:
~~~~~~~~~~~~~~~~~~~
The ModemLink device still needs a lot of work. There are a bunch of little things I'd like to add (Flags & Tags). Some of the more important enhancements I'd like to include one day are:
1) Redesign of the protocol handler task. As it is now it is pretty darn ugly. I would like to re-write it except this time base it on a cellular automaton.
2) Add at least one more protocol. Something like the Sliding Window

protocol which is much nicer for high through-put communications.
3) More commands for the device to handle (like CMD_Flush).
4) Make it possible to send a linked list of packets in a single write
request to the device.
5) Add support for Sockets (if you check out the LinkPkt struct you'll
see it has a Socket field already - but is not used).
6) Allow for shared Units
Currently my plans are to switch projects and start working on something
new and exciting. However, if there is enough interest I may continue
working on the ModemLink project. Please send me all your feedback.

1.16 ModemLink device - Release History

Release History:
~~~~~~~~~~~~~~~
R1.2 V36.2 (Oct 26 1997):
- ModemLink.device would not create the ModemLink Semaphore which is needed
to make sure that only one instance of modemlink.device is running for a
given Unit. This has been fixed.
R1.1 V36.1 (Oct 25 1997):
- SendModemCMD now checks to see if modem is OFF only if issued command is
never echoed back from modem. This is to be compatible with newer
modems which auto power-up when first command is issued.
- Cleaned up some documentation.
- Changed the scoptions file so that it now sets PARAMS=BOTH. It will now
use both registers AND stack for parameters.
- Added StormC pragma file in include/StormC_pragmas.
R1.0 V36.0 (May 10 1997):
- Initial release
R0.0 V00.0 (July 4, 1996):
- Modemlink Project begins

1.17 ModemLink device - About Michael Veroukis...

About The Author:
~~~~~~~~~~~~~~~~
I've owned an Amiga 500 since 1987, and I bought one of the very first
Amiga 1200s to come out. I still love this little computer, and still have
fun developing on it. My dream has always been to be a real life commercial

developer for the Amiga. Unfortunately, this dream seems very unlikely to
be fulfilled due to the current situation the Amiga is in. But I'm still
being hopeful that Amiga will make a comeback.
I have a Degree in Computer Science from the University of Manitoba. I've
been working at SHL Systemhouse Inc. for almost a year now doing Sybase SQL
and PowerBuilder work. The pay is good, the work is good too, but I know
where I'd rather be... (Is Interplay hiring?!? ;-)
So, I write Amiga software when ever I get some free time at home. The
ModemLink device was actually written for a friend of mine who's writing
a pretty cool sport/fantasy/strategy game called FantasyBowl. I felt that
his game should be modem playable, but he felt that adding modem playability
would take too long. So, I told him I'd help. :)
Now that I'm done (or so I think) I want to get back to work on another
project I've been working on; Bala. This is a game I've been developing for
quiet some time now. It will be AGA only (perhaps Cyber GFX as well), and I
hope, lots of fun. It's hard to say when I'll have it finished, but it won't
be any time soon.
...
If you have any comments, suggestions or bug reports, please contact me.
To do so you have a few of options:
My e-mail address is: yogi@autobahn.mb.ca
My web page is at: www.autobahn.mb.ca/~yogi
Snail mail address: Michael Veroukis
1000 Radisson ave.
Winnipeg MB
Canada R3T 1R3

1.18 ModemLink device - Special Thanks

Special Thanks:
~~~~~~~~~~~~~~~
First of all, I'd like to thank my friend Ken Paulson who gave me the idea
and motivation to make this thing, not to mention all those exciting
arguments (which I always won of course ;).
Thanks also go out to James Ceraldi (Aurora Works) who helped me test
modemlink and provided a few suggestions and bug reports.
I'd also like to thank all those weird and wacky C= engineers that made
the Amiga the cool computer that it is.
~~~~~~~~~~~~~~~

1.19 ModemLink device - Example 1

```
/*
** EXAMPLE 1:
**
** This is a very simple example of how to open the modemlink device and
** get a pointer to the base address of the device. Getting the base
** address is important because it is neccesary when making direct function
** calls to the device. Of course this is not neccesary when using the
** linked library version of modemlink.
*/
#include <exec/types.h>
#include <exec/io.h>
#include <proto/exec.h>
#include <proto/dos.h>
#include <ModemLink/ModemLink.h>
#include <proto/ModemLink.h>
struct Library *ModemLinkBase;
void main(int argc, char **argv)
{
struct IOExtLink *LinkIO;
struct MsgPort *LinkMP;
BYTE error;
if (LinkMP = CreateMsgPort()) {
if (LinkIO = CreateIORequest(LinkMP, sizeof(struct IOExtLink))) {
/* Here's where we open the device */
if (!(error = OpenDevice(MODEMLINKNAME, 0L,
(struct IORequest *)LinkIO, 0L))) {
/* Here's were we get the base address of the device */
ModemLinkBase = (struct Library *)LinkWriteIO->IOLink.io_Device;
...
/* And here's where we close the device */
CloseDevice((struct IORequest *)LinkIO);
}
DeleteIORequest((struct IORequest *) LinkIO);
}
DeleteMsgPort(LinkMP);
}
}
```

1.20 ModemLink device - Example 2

```
/*
** EXAMPLE 2:
**
** This is a full blown example of how the entire modemlink system works.
** This can be found in the Examples directory, but has been included here
** for easy reference.
*/
#include <exec/types.h>
#include <exec/memory.h>
#include <exec/io.h>
#include <devices/serial.h>
#include <utility/tagitem.h>
#include <proto/exec.h>
#include <proto/dos.h>
#include <proto/intuition.h>
#include <stdio.h>
#include <string.h>
#include <ModemLink/ModemLink.h>
#include <proto/ModemLink.h>
#include "devicestuff.h"
struct Library *ModemLinkBase;
void main(int argc, char **argv)
{
struct IOExtLink *LinkWriteIO, *LinkReadIO;
struct IOExtSer *SerIO;
struct MsgPort *LinkWriteMP, *LinkReadMP, *SerMP;
char buf[512];
int Connect, BusyCount = 0;
printf("Test ModemLink Device -- Let's hope this thing works!!!\n");
if (argc < 3) {
if (LinkWriteMP = CreateMsgPort()) {
if (LinkWriteIO = CreateIORequest(LinkWriteMP, sizeof(struct IOExtLink))) {
if (!(Connect = OpenDevice(MODEMLINKNAME, 0L, (struct IORequest *)LinkWriteIO, 0L))) {
ModemLinkBase = (struct Library *)LinkWriteIO->IOLink.io_Device;
if (OpenSerialDevice(&SerMP, &SerIO, "serial.device", 0L)) {
if (argc == 2)
do {
```

```
if (BusyCount)
Delay(150);
printf("Dialing %s....\n", argv[1]);
Connect = ML_DialTags(SerIO, argv[1], TAG_DONE);
printf("Dialer ReturnCode: %d\n", Connect);
} while (Connect == MODEM_BUSY && BusyCount++ < 2);
else {
printf("Waiting for incomming call...\n");
Connect = ML_AnswerTagList(SerIO, NULL);
}
printf("Modem ReturnCode: %d\n", Connect);
if (Connect == MODEM_CONNECT) {
Connect = ML_EstablishTags(LinkWriteIO, SerIO, TAG_DONE);
printf("Establish ReturnCode: %d\n", Connect);
if (Connect == EstErr_OK) {
printf("Connected!!!\n\n");
printf("Type message and hit return to send\n");
printf("Hit return to check for incomming messages\n");
printf("Enter '.' and hit return on a new line to quit\n\n");
if (CloneIO((struct IORequest *)LinkWriteIO, &LinkReadMP, (struct IORequest **)&LinkReadIO)) {
LinkReadIO->IOLink.io_Command = CMD_READ;
LinkReadIO->IOLink.io_Data = 0;
SendIO((struct IORequest *)LinkReadIO);
while (1) {
printf("\n: ");
gets(buf);
if (buf[0] == '.' && buf[1] == 0)
break;
if (buf[0] > ' ') {
printf("Sending: [%s]\n", buf);
LinkWriteIO->IOLink.io_Command = CMD_WRITE;
LinkWriteIO->IOLink.io_Data = &buf;
LinkWriteIO->IOLink.io_Length = strlen(buf) + 1;
DoIO((struct IORequest *)LinkWriteIO);
}
if (CheckIO((struct IORequest *)LinkReadIO)) {
WaitIO((struct IORequest *)LinkReadIO);
DisplayBeep(NULL);
if (!LinkReadIO->IOLink.io_Error) {
```

```
printf(">> [%s]\n", LinkReadIO->IOLink.io_Data);
FreeMem(LinkReadIO->IOLink.io_Data, LinkReadIO->IOLink.io_Length);
LinkReadIO->IOLink.io_Command = CMD_READ;
LinkReadIO->IOLink.io_Data = 0;
SendIO((struct IORequest *)LinkReadIO);
}
else
printf("io_Error: %X\n", LinkReadIO->IOLink.io_Error);
}
LinkWriteIO->IOLink.io_Command = MLCMD_QUERY;
DoIO((struct IORequest *)LinkWriteIO);
if (LinkWriteIO->IOLink.io_Error)
break;
}
if (!LinkReadIO->IOLink.io_Error) {
AbortIO((struct IORequest *)LinkReadIO);
if (!CheckIO((struct IORequest *)LinkReadIO))
WaitIO((struct IORequest *)LinkReadIO);
}
DeleteIO_MP(LinkReadMP, (struct IORequest *)LinkReadIO);
}
ML_Terminate(LinkWriteIO);
}
}
SafeCloseDevice(SerMP, (struct IORequest *)SerIO);
}
CloseDevice((struct IORequest *)LinkWriteIO);
}
else
printf("ERROR: Could not open modemlink.device\n");
DeleteIORequest((struct IORequest *) LinkWriteIO);
}
DeleteMsgPort(LinkWriteMP);
}
}
else
printf("\nUSAGE: TestMLDev <PhoneNumber>\n");
}
```

1.21 ModemLink device - Example 3

```
/*
** EXAMPLE 3:
**
** This routine can be used to make a copy of any IORequest block.
** It will create a new MsgPort and new IORequest struct. It will then
** copy the contents of the previously created IORequest struct over the
** new one. It must then make the new IORequest struct point to the new
** MsgPort struct. Keep in mind that the previously allocated IORequest (the
** first parameter) must have been initialized by OpenDevice().
*/
int CloneIO(struct IORequest *IO,
struct MsgPort **NewMP,
struct IORequest **NewIO)
{
if (IO && NewMP && NewIO) {
if (*NewMP = CreateMsgPort()) {
if (*NewIO = CreateIORequest(*NewMP, IO->io_Message.mn_Length)) {
CopyMem(IO, *NewIO, IO->io_Message.mn_Length);
(*NewIO)->io_Message.mn_ReplyPort = *NewMP;
return(1);
}
DeleteMsgPort(*NewMP);
}
}
*NewMP = 0;
*NewIO = 0;
return(0);
}
```