- Copying disks - and using the trackdisk.device by Bob Rakosky
- An early glimpse of 1.4 by Mike Todd
- A Maker Of Islands - landscapes and fractals by Chris Reynolds & Danny Ross
- Programming for MIDI in Modula 2 by Dan Brookshier
- What does 'Freely Distributable' really mean? by Steve Ahlstrom
- Assembly Language Programming - part 3 by Jim Butterfield
- Interrupts and the Amiga by Ian Potts
- The Nottingham ADAB conference - a report by Mike Todd
- Plus regular columns by Larry Phillips, Don Curtis and Steve Ahlstrom

Two FracScapes

Registered by Australia Post Publication No. NBP9435
Briwall Australia is a new wholly Australian owned Mail Order Software house, established to give Commodore computer users access to the same wide range of books, peripherals and software as their counterparts in the U.S.A. and Europe.

By linking up with Briwall, a major American mail order/software publishing house, we are able to supply the full range of Amiga products at prices which are extremely competitive. Another benefit of the bond with Briwall in America is access to their Technical Assistance Section to assist clients who may be having difficulty in getting their programs up and running.

Detailed below is a cross section of some of our products and prices for Amiga programs and books etc.

<table>
<thead>
<tr>
<th>BOOKS</th>
<th>PRODUCT</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001 Things to Do with Amiga</td>
<td>$25.00</td>
<td></td>
</tr>
<tr>
<td>Amiga C Beginners</td>
<td>$30.00</td>
<td></td>
</tr>
<tr>
<td>Advanced Amiga Basic</td>
<td>$30.00</td>
<td></td>
</tr>
<tr>
<td>Machine Language Programming Guide</td>
<td>$35.00</td>
<td></td>
</tr>
<tr>
<td>Amiga Tips &amp; Tricks</td>
<td>$30.00</td>
<td></td>
</tr>
<tr>
<td>Amiga DOS Express</td>
<td>$43.00</td>
<td></td>
</tr>
<tr>
<td>Inside Amiga Graphics</td>
<td>$35.00</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EDUCATIONAL</th>
<th>PRODUCT</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Animal Kingdom</td>
<td>$55.00</td>
<td></td>
</tr>
<tr>
<td>Fraction Action</td>
<td>$75.00</td>
<td></td>
</tr>
<tr>
<td>Ghostly Grammar</td>
<td>$75.00</td>
<td></td>
</tr>
<tr>
<td>Linkword—French</td>
<td>$43.00</td>
<td></td>
</tr>
<tr>
<td>Linkword—German</td>
<td>$43.00</td>
<td></td>
</tr>
<tr>
<td>Linkword—Italian</td>
<td>$43.00</td>
<td></td>
</tr>
<tr>
<td>Linkword—Spanish</td>
<td>$43.00</td>
<td></td>
</tr>
<tr>
<td>Mavis Beacon Typing</td>
<td>$75.00</td>
<td></td>
</tr>
<tr>
<td>Planetarium</td>
<td>$100.00</td>
<td></td>
</tr>
<tr>
<td>Surgeon</td>
<td>$72.00</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CREATIVITY/GRAPHICS</th>
<th>PRODUCT</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deluxe Paint III</td>
<td>$229.00</td>
<td></td>
</tr>
<tr>
<td>Deluxe Print II</td>
<td>$158.00</td>
<td></td>
</tr>
<tr>
<td>Digiview Gold</td>
<td>$319.00</td>
<td></td>
</tr>
<tr>
<td>Draw 2000</td>
<td>$375.00</td>
<td></td>
</tr>
<tr>
<td>Media Font Disks</td>
<td>$30.00</td>
<td></td>
</tr>
<tr>
<td>Media Animation Backgrounds</td>
<td>$30.00</td>
<td></td>
</tr>
<tr>
<td>Media Clip Art</td>
<td>$30.00</td>
<td></td>
</tr>
<tr>
<td>Page Flipper</td>
<td>$80.00</td>
<td></td>
</tr>
<tr>
<td>Printmaster Plus</td>
<td>$58.00</td>
<td></td>
</tr>
<tr>
<td>Ultra Cad</td>
<td>$358.00</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GENERAL BUSINESS</th>
<th>PRODUCT</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acquisition V1.3</td>
<td>$429.00</td>
<td></td>
</tr>
<tr>
<td>Professional Draw</td>
<td>$286.00</td>
<td></td>
</tr>
<tr>
<td>Data Retrieve</td>
<td>$105.00</td>
<td></td>
</tr>
<tr>
<td>Easy Ledgers</td>
<td>$422.00</td>
<td></td>
</tr>
<tr>
<td>Excellencel</td>
<td>$369.00</td>
<td></td>
</tr>
<tr>
<td>Kindwords</td>
<td>$129.00</td>
<td></td>
</tr>
<tr>
<td>Microfiche Filer Plus</td>
<td>$256.00</td>
<td></td>
</tr>
<tr>
<td>Phasar V3</td>
<td>$129.00</td>
<td></td>
</tr>
<tr>
<td>Page Stream</td>
<td>$269.00</td>
<td></td>
</tr>
<tr>
<td>Pen Pal</td>
<td>$215.00</td>
<td></td>
</tr>
<tr>
<td>Superbase Prof. V3</td>
<td>$475.00</td>
<td></td>
</tr>
<tr>
<td>Word Perfect</td>
<td>$459.00</td>
<td></td>
</tr>
<tr>
<td>Works—Platinum Edit.</td>
<td>$420.00</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ENTERTAINMENT</th>
<th>PRODUCT</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>4x4 Off Road Racing</td>
<td>$59.00</td>
<td></td>
</tr>
<tr>
<td>Heroes of the Lance</td>
<td>$59.00</td>
<td></td>
</tr>
<tr>
<td>Arctic Fox</td>
<td>$30.00</td>
<td></td>
</tr>
<tr>
<td>Battle Chess</td>
<td>$64.00</td>
<td></td>
</tr>
<tr>
<td>Bridge 5</td>
<td>$50.00</td>
<td></td>
</tr>
<tr>
<td>Chessmaster 2100</td>
<td>$55.00</td>
<td></td>
</tr>
<tr>
<td>Destroyer</td>
<td>$60.00</td>
<td></td>
</tr>
<tr>
<td>Falcon</td>
<td>$56.00</td>
<td></td>
</tr>
<tr>
<td>Flight Simulator 2</td>
<td>$72.00</td>
<td></td>
</tr>
<tr>
<td>Rocket Ranger</td>
<td>$58.00</td>
<td></td>
</tr>
<tr>
<td>Roger Rabbit</td>
<td>$58.00</td>
<td></td>
</tr>
<tr>
<td>Virus</td>
<td>$45.00</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UTILITIES</th>
<th>PRODUCT</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC Basic</td>
<td>$279.00</td>
<td></td>
</tr>
<tr>
<td>Arexx Language</td>
<td>$72.00</td>
<td></td>
</tr>
<tr>
<td>Assemble</td>
<td>$143.00</td>
<td></td>
</tr>
<tr>
<td>CLI Mate</td>
<td>$58.00</td>
<td></td>
</tr>
<tr>
<td>Developers Pak 2</td>
<td>$143.00</td>
<td></td>
</tr>
<tr>
<td>Grabbit</td>
<td>$42.00</td>
<td></td>
</tr>
<tr>
<td>Mac Emulator</td>
<td>$215.00</td>
<td></td>
</tr>
<tr>
<td>Power Windows 2.5</td>
<td>$129.00</td>
<td></td>
</tr>
<tr>
<td>Raw Copy 1.3</td>
<td>$85.00</td>
<td></td>
</tr>
<tr>
<td>Quarterback</td>
<td>$95.00</td>
<td></td>
</tr>
<tr>
<td>Workbench V1.3</td>
<td>$43.00</td>
<td></td>
</tr>
</tbody>
</table>

To order any of the above programs or obtain our full catalogue, please write to the address opposite:
With Winter now behind us, and tax cheques in our pockets, we can expect to have all sorts of products dangled before us with exhortations to go out and buy.

Commodore will be in there, too. Each year about this time, the drive towards the lucrative Christmas market starts in earnest, and this year should not prove any exception. But there is something just a little different about this year, by my reading of the tea leaves.

Let's step back a little, and look at the events leading up to the present time. In the USA, Max Toy was replaced earlier this year because he failed in his quest for increased penetration of the US market (the larger portion of Commodore sales occur outside the continental USA). In Oz, we too have a new leader, Pat Byrne. But the products they have to sell, the C64, PC and Amiga, have fallen behind the current technology in the marketplace. Surprised? Well, its true. PCs at 25 and 33 MHz are seen more frequently now. The Mac is selling so well that production can hardly keep up. And the Amiga? What's new there? It keeps rumbling along at 7 MHz, and it is probably being outsold in many quarters.

Commodore simply has to do well this season. If not, then the company stocks (already suffering in the tough world out there) will fall. And there's no lonelier place than the lower quadrant of the sales graph.

Maybe there is something new and exciting out there, but to judge from Commodore's promotion (or should that be non-promotion?) of the Amiga, the average buyer would have difficulty in finding out. Our own case is one in point. In our twelve months of operation, we have received only one set of news from the CBM public relations camp. One! And that wasn't much use, anyway. Commodore hasn't responded to our offer of space to keep you informed. What do we have to do?

Maybe my perception of the job of a public relations company is not accurate, but their traditional role in our society has been to ensure that news and views are spread right around to all interested parties, and to foster an appreciation and value of what its client is doing. As it was put to me by a colleague, we have to approach them! Well, we will pass on that one. We're busy enough as it is.

We all have a stake in Commodore, and Commodore products. We want to know about their products and share in their successes. Sad to relate, they give the impression of not caring about their users.

For this issue we have teamed up with Tim Strachan from MegaDisc to bring you a consolidated index of MegaDiscs 1 to 12 - look in the middle of this issue. This is a short-form index, designed for quick reference. Tim will carry a more detailed index on a future disk, so those of you who want can print out the 30-odd pages it will probably take. We didn't have that much space.

Paul Blair

MAIL TO: OR PHONE:
TRANSACTOR (AUST) PTY LTD (062) 88 3584
35 CALDER CRESCENT 7-9pm weeknights
HOLDER ACT 2611

Please send me 6 issues of TRANSACTOR for the AMIGA
at $29 per subscription ($A35 for New Zealand)
And (just for good measure)
Magazine backorders Vol 1 . . .
Vol 2 . . .
Extra copies of Vol 2 . . .
Transactor for the Amiga disks at $11 each

Circle the disks or magazines you want. Please add $2 p+p to disk orders and magazine backorders.

Hurry all this to me at:
Name .................................................................................................................................
Address ..............................................................................................................................
................................................................................................................................. Postcode

If paying by Bankcard/Visa, my number is .................................................................
Expiry ................................. Phone ................................. Signed .................................

CHECK THE DETAILS, THEN MAIL TO 35 CALDER CRESCENT HOLDER ACT 2611
From the Kernel
by Mike Todd
A missing bit of code (Oops!) ... new documentation ... Word Perfect - first it's dead, then it's not ... reveal the beauty of
Base ... the ultimate disassembler? ... and, coming soon, the ultimate assembler?

ViewPort
by Larry Phillips
The idea of a standard user-interface is admirable - but does it represent a move away from the ideal programming en-
vironment? Does its very existence mean unnecessary restrictions on programmer and user alike? Larry starts an inter-
esting debate.

Dispatches
by Don Curtis
Don investigates the ripple effects caused by small changes to software, and explains why compromises often mean the
appearance of products that we wouldn't otherwise see. In fact, there's probably a darn good reason for all those irri-
tating short-falls in a product. Before he goes, Don also discusses the philosophy and how and where the HELP key
should act.

Access - the best of non-commercial software
by Steve Ahlstrom
What is the difference between Shareware and freeware? Just what does 'freely distributable' really mean? What
exactly is the 'Public Domain'? Steve looks at these issues, and discovers that many authors unwittingly give away all
rights to their software simply because they don't understand that by placing their software in the public domain, they
give away all rights to that software! Steve also continues his examination of the collection of Freely Distributable
software and tries out a keymap editor, a couple of system bug-fixes and a few utilities.

And now ... 1.4!
by Mike Todd
With the emergence of the new extended chip set for the Amiga, Commodore is in the process of producing a major
upgrade of the operating system. Some prefer to call it 2.0 because of the scale of the upgrade, but Commodore still
refer to it as 1.4. Here is the latest news of what Commodore hopes will be some of the major features of the new
operating system.

Cover Pictures: Two FracScapes
by Danny Ross & Richard Farrow
The two cover pictures are examples of landscapes generated by AMOI, A Maker Of
Islands, the program described in this issue. They are created from contour maps (shown
in the insets) with the surface detail generated using the fractal technique of mid-point
displacement.

The top picture (by Richard Farrow) is a straightforward IFF picture generated by the
program, while the bottom picture (by Danny Ross) has had some 'post-processing'.

Publisher
Paul Blair

Editor
Mike Todd

Advertising Manager
Harry Gemmell

Contributing Writers
Steve Ahlstrom
Scott Ballantryne
Peter Booth
Dan Brookshire
Anthony Bryant
Jim Butterfield
Bettty Clay
Don Curtis
Andy Dawson
John Faichney
Leonardo Fei
Brad Fowles
Eric Giguère
Cathryn Graham
Derek Grime
Dr. Rahman S. Haleem
Charlie Heath
Graeme Ing
Chad Innes
Amanda Jones
Louis Markoya
Bryc¥ Nesbitt
Dr. Howard Oakley
Rob Peck
Larry Phillips
Ian Potts
Chris Reynolds
Danny Ross
Eric Salter
Dan Schein
Steve Simpson
John Toebes
Victor A. Wagner
Clark Williams
David Wood
Assembly Language, Part 3
by Jim Butterfield

As Jim continues his exposition of machine language programming, he examines the concept of relocatable code, and the function of the 'hunks'. He then shows how to write 'pure' code and illustrates this with a simple file reading program - one version in an impure state, and the other a pure version that can be made resident.

The U.K. Developers' Conference
by Mike Todd

A report on the conference held at Nottingham University over the weekend of 15th & 16th July 1989. This conference was organised by developers, for developers. With speakers from the U.K. and the U.S.A., it proved to be a useful learning experience for all who attended.

The Amiga Interrupts
by Ian Potts

Even when it's not doing anything, the Amiga is a very active machine. Interrupts come from all sides, and perform every activity from detecting the mouse position to controlling the screen display, from handling RS232 data incoming to playing audio samples. Ian looks at the Amiga's highly organised interrupt hierarchy, and shows how to write and install interrupt driven code with two examples - one moves a user-sprinked around the screen, the second makes a noise.

AMOI - A Maker Of Islands
by Chris Reynolds & Danny Ross

The 'islands' in the title are landscapes, and the 'maker' is a program which takes contour maps and creates perspective IFF pictures of the landscape. Fractals are undoubtedly the 'in-thing' in the world of mathematical-graphics, and Chris and Danny have used the concept of 'mid-point-displacement' to enhance their 3D projections of the contours drawn by the user. Unfortunately, the code for the program is too long for the magazine, but is available on the disk that complements this issue. An executable version is also available on that disk - but look out for an enhanced version in the future.

MIDI - the software
by Dan Brookshire

Having looked at the hardware behind MIDI, and some simple software techniques, Dan goes one step further and describes some programming techniques for handling MIDI data, using Module 2. Dan also describes the data structure of MIDI messages, and explains what to do with the data when it arrives.

Disk copying
by Bob Rakosky

The trackdisk device is at the heart of the Amiga's disk operations. Bob looks at how to access the disk drive at this low level, and in the process develops a useful disk copying utility that takes advantage of the available RAM in the Amiga. If there is sufficient memory to hold an entire disk, then the program allows multiple copies to be made from a single disk read.
From The Kernel

There’s been a lot happening since the last issue, which we brought out slightly ahead of schedule in order to be out in time for the Commodore show. For programmers of the Amiga, perhaps the most significant event has been the Developers’ Conference, held at Nottingham University in July. There is a report of this on page 29, with the latest news on 1.4 on page 18.

Sorry...

Okay ... so who spotted the little bit of listing missing at the end of Jim’s Hunker program on page 23 of TransAmi, Volume 1 Issue 5? Well at least one of you did!

I can only apologise for letting the auto-listing-formatter get the better of me; the missing piece of code is at the end of this editorial.

The new reference books

Now that we have 1.3 in wide circulation, Commodore and Addison-Wesley have started to release updated Amiga reference books.


These books are very expensive, but really are the definitive reference guide and are much revised versions of the originals.

For instance, the new Includes & Autodocs manual is very thorough, covering all the include files, and Kernel routine descriptions available in release 1.3 - it also has the full IFF documentation and there are reference charts of hardware registers, assembly prefixes (like _IV_ and _IOEXTSER_), C language include definitions and, ideal for ROM hackers everywhere, cross-referenced structure and library offset tables.

A new 1.3 DOS manual from Bantam can’t be expected until Commodore digs itself out of the hole it created when it gave Bantam sole and exclusive rights to the contents. Commodore is in the crazy position of having sold the rights to its own documentation! Attempts are being made to recover those rights, and it doesn’t seem that Commodore will give Bantam any of the 1.3 update information until the exclusivity of the deal is revoked.

However, Commodore can’t publish the updates itself since these are covered by the agreement! Isn’t it about time Commodore learnt not to point a loaded gun at its own feet?

The Word Perfect saga

Many of those who keep their ears to the ground will know that, back in April, the Word Perfect Corporation announced that all new development work on the Amiga had been stopped.

This came as a huge shock to all Word Perfect users worldwide. WP Corp. had declared that they would finish version 4.1, and continue to fix bugs, but no new development work was to be done - to all intents and purposes, Word Perfect would be a dead product.

It would appear that Amiga sales failed to pay its way - even though development costs had long since been recouped.

Reaction on the networks was swift and vociferous. Having posted the news, one of the board of directors continued to monitor reaction and in May, the decision was made to continue with Amiga development after all.

The product will continue to be enhanced over the next 18 months, aiming towards a version 4.2. At the same time, Word Perfect Corporation would monitor the situation carefully and the future of the Amiga in their eyes would depend on sales.

ExecDis

Now, far be it from me to encourage hackers to dig into the ROM code ... but that’s just what I’m about to do!

The exec.library is written in assembler, and is a good example of well-written code. It is worth looking at - but it will take you a long time to find your way around.

Markus Wandel has produced a full disassembly of the code for you - and provided detailed comments into the bargain. Unfortunately, the code is copyright by Commodore-Amiga and so can’t be distributed (although isn’t there something in the copyright laws about being able to publish extracts?); but Markus has produced an elegant solution to this problem.

If you have 1.2 in your system, follow the instructions which come with ExecDis (available in many repositories of freely
distributable software, but also on this issue's TransAmi disk for convenience) and, using the supplied disassembler and 'script', you will be the proud owner of 125 pages of well commented disassembly running from $FCD000 to $FC34CB.

I've had the opportunity to compare some of the resulting disassembly against original source code - and, apart from missing labels and symbols, it is actually better commented than the original!

ReSource

If you're hooked on disassembling the system, just enjoy browsing through other people's programs, or want to have a closer look at some of your own code, you could do a great deal worse than get ReSource, by Glen McDiarmid.

This is a commercial product which gives very powerful disassembling features. It knows all about system symbols, library offsets and the like and can even make intelligent guesses as to what is code and what is data.

It is an enormously powerful tool, and could be described as an 'intelligent disassembler'. While the human interface is not exactly perfect, intelligent use of the powerful macro capability more than makes up for it.

We published a review back in May (TransAmi Volume 1, issue 6), and since then the program has undergone substantial revision.

A demonstration version is included on the TransAmi disk for this issue and, if you wish to buy a full-featured version (it costs around £40), get in touch with us here at Transactor UK.

Coming soon ....

'Is your assembler slowing down your program development cycle by making you wait?'

So starts the leaflet on a brand new assembler package from Argonaut Software. Yes ... I did say Argonaut; the boys that brought you mega-fast games now bring you a mega-fast assembler. The Argonaut Assembler is called ArgAsm.

Jez San claims that the new assembler out-performs the three major assemblers in use today by being many times faster - with DevPac 2.0, for instance, a complete assembly of a simple file full of RTS5 or NOE5 proceeds at around 75,000 lines per minute. ArgAsm performs at 1,000,000 lines per minute!

In a more realistic test, with real code, ArgAsm was over three times faster (and up to ten times faster) than DevPac.

A optional built-in editor (ArgEdit) is also available, with full multi-file, multi-window support - and, when you're ready, just hit the assemble key and the current window will be assembled with error messages brought back into the editor for convenience.

Several novel features have been added to the assembler, including instruction cycle timings in the listing file, the ability to see which instructions affect the status flags and help in error messages regarding possible addressing modes. It also includes the usual instruction optimisation, although care has been taken to avoid those which could damage your code - and each individual optimisation can be turned off.

Despite all the features, ArgAsm supports a complete set of directives formed from a superset of DevPac, Cape and Assem - so existing files can be assembled with little or no change.

Well, I'm naturally sceptical about such an apparently ideal software product and TransAmi will be reviewing ArgAsm as soon as it is available - which is just about any day now.

For further details, contact Argonaut Software on 01 906 3744.

Mike Todd

The missing piece of Jim Butterfield's hunker program
ViewPort

A system-standard file-requester ... or not?

by Larry Phillips
Copyright © 1989 Larry Phillips

'I'd like some strawberry ice cream please.'

'Sorry, we only have vanilla.'

'No choice?'

'No, but vanilla is the best flavour. Absolutely the only flavour you need. It is cold, a nice yellowish white colour, and won't offend the taste buds like some flavours. And best of all, it makes choosing easy.'

'But what if I don't like vanilla?'

'But everyone likes vanilla. That's why we serve it. If you really don't like it, you could make your own with our handy ice cream maker.'

Ice cream in a computer magazine? That's it, you say, Phillips has really cracked up now!

Does the conversation ring a bell? I was going to make the ice-cream apple flavoured, but that would have been too obvious. The unlikely scenario above is how I perceive the mentality of those who would impose their world upon us under the guise of a 'standard user interface'.

I wouldn't mind so much if the proponents of a standard user interface were to stop short of the details of the implementation. They don't though. They start talking about whether or not a file requester should sort while loading, sort on demand, or not sort at all. They would dictate to us what the gadgetry looked like, whether double-clicking on a filename should be the same as clicking on the OK or LOAD gadget, whether you should select the parent directory with a gadget or as part of the file list. They would dictate the smallest detail, all in the name of a standard user interface.

It is frustrating at times to talk to those who hold the Mac interface near and dear to their hearts. It isn't the fact that they like a standard user interface that bothers me, because I agree totally that a standard is desirable, but the fact that they come from a background where it is considered acceptable for one person, or group of persons, to dictate what that standard should be, right down to the most niggling little details.

Those programmers that feel the standard isn't quite what they want in their program are, of course, free to write their own user interfaces. Whether the program that results from this will sell or not is another thing altogether.

There is a fair amount of pressure from Apple, from Mac users and from other Mac developers against any changes to the user interface. Still, the programmer is certainly within his rights to make his program behave in any way he sees fit.

There are many problems inherent in this situation, and though it does have its good points, I feel that there is no reason we cannot take the good points without the problems, and apply them to the Amiga.

A common wish among Amiga owners is for a file requester that is built in to the system, so I will use this as an example of what might be done to promote a standard, while eliminating the problems.

The primary benefit of a standard user interface is that the new user can easily pick up any application and immediately know that it operates, for the most part, just like all his other applications. This reduces FUD (Fear, Uncertainty, and Doubt), and allows both novice and veteran alike to quickly become familiar with a new program.

The problems really boil down to one thing. If you, as a user of the machine, don't happen to like the user interface, or any part of it, that's too bad. You are stuck with what you get. If a programmer decides that he wants to write his own file requester, and you buy the program, you are stuck with that too. The only difference being that in the case of the system supplied file requester, you are stuck with the offical version, and in the second case, you are stuck with the programmer's version.

The first questions we might ask are:

'Who is the standard for?'

'Can we figure out a way to have a standard and at the same time, have choices?'
The answer to the first question, in my opinion, is that the standard is for the user. That it makes things easier for the programmer to have a built-in file requester is also a good thing, but in the final analysis, the benefits are to the user.

The answer to the second question is a resounding ‘Yes!’ and it is this that I want to address in the following proposal. Bear in mind that I am making the assumption that CM will provide, in some future release, a built-in file requester [possibly in 1.4 - ED], and that these comments are meant to address a possible implementation of it. This proposal does not concern itself with the details of how the user interacts with the file requester. It does not deal with the look, feel, number of gadgets, or size. It deals only with the interface between the calling program and the file requester itself, and you will see why in a moment.

The general idea is that a system-provided file requester should be configurable in some ways, through the use of user settable flags. It should be callable by any program, easily, much as any other system routine is callable. It should provide all the information the user would expect from a file requester, which is to say that it should provide a path and a file name (or multiple file names) to the calling program.

The file requester should also be replicable, easily, by the end user. Consider the implications of replicability. A programmer, given an easily callable file requester that provides adequate functionality, would be hard pressed to come up with a reason to roll his own.

The only reason programmers of Mac and Presentation Manager applications even consider doing it is because they feel that the file requester provided by the system is inadequate, ugly, or both.

Any inadequacy can be overcome by providing a calling convention that is both complete and extensible. Any ugliness can be overcome by allowing easy replacability. Completeness of the calling convention is the most difficult part of this scheme when you consider that the convention must allow for all ‘proper’ parameters or flags likely to be needed, and for both now and in the future.

Programmers being what they are, would undoubtedly come up with a flag or parameter they simply must have, just as the ROMs are in full production and ready to ship. The second part, the facility to allow easy replacement of elements of the system are already in place, with fonts, keymaps, printer drivers, device drivers, libraries, and even individual routines within libraries.

Let’s start with the programmer’s needs. An application might need to be able to specify that it can accept multiple filenames, or that it can only handle one. He needs to be able to provide ‘title text’ to label the requester, a path name for the directory listing, whether he wants the requester listing to filter the directory to show only files matching a certain pattern (or only files not matching a certain pattern), and probably a few I can’t think of at this moment. He needs to be informed if a selection was made, if it was a multiple selection, if the requester was cancelled without a selection, and of course, the path and file names themselves.

The programmer need not concern himself with what the requester looked like, whether it sorted a directory, or whether mouse or keyboard or voice input was used to make the selection. Those are best left up to the user.

Configurable options, perhaps set as an option within preferences or a similar program, might include such things as ‘hot’ selection ability, double-click or OK gadget only, whether it comes up in a fixed position or in relation to the mouse, and so on. Options settable in this manner would be those that are of no particular concern to the programmer, and are more a matter of taste for the user. The setting would ideally affect any file requester.

This brings up the heart of the matter. The programmer has his system call, and can depend on the results of that call. If the file requester can be replaced easily, the user has his choice of file requesters. In this way, we can have a standard in more ways than one.

The first standard is that the system file requester is what the new Amiga owner will see. It will be called for most programs, the only exceptions being the older programs that use their own scheme, and older programs that have yet to be updated to call it. It will provide a common ground for talking about and learning the operation of the system as well as the individual applications.

The other standard is obtained when a user decides to replace the system-supplied file requester with one that he prefers. The replacement, while it can in one way be looked at as non-standard, really is a standard after all, if it is called whenever the system requester would have been called.

Thus, all applications calling in the normal manner will call the same file requester without knowing or caring that a replacement is being called, making all applications using it act identically from the user’s viewpoint, and in a way that the user prefers it to act. It seems to me that all the advantages of fixed system file requesters are obtained, while still allowing for choice on the part of the user.

I think that choices are important, since every Amiga owner seems to have a preference for one file requester over another. Personally, I like one that can be dragged around the screen and is small enough to allow me to see any text that might otherwise be hidden. Others prefer a large one with the ability to display a lot of file names. Some like the ability to use the keyboard, or the mouse, or both. Do you like gadgets for selecting a volume name, or do you prefer a textual listing? Do you want assigned directories to show up along with the volume names?
All these preferences are things that need not concern the programmer, who can rest assured that when he calls for a filename, the user will be supplying a filename in the way that is most comfortable for the user, rather than in a way left up to Commodore or the application.

We need not limit the replacement to any particular appearance; indeed it need not appear on screen at all. A sight-impaired person might want a large file requester, even if only three or four files may be seen at a time. A blind user might want a talking file requester that has no visual representation at all. In fact, almost any conceivable method of presenting a list of files could be handled by a replacement.

A terminal hooked up to a serial port, might make good use of a file requester presented in a traditional numbered menu display, allowing the user to choose via keystrokes without having to run across the room or across town to make the selection at the Amiga’s keyboard.

Remember, though I am talking about a file requester, the principle could be applied to other user interface building blocks. I think we can have standards and still allow creativity. We can have an easy to learn machine that can be changed as the user advances. We can have a machine that is unparalleled in its accommodation of widely varying preferences or needs.

So, who has some great ideas for a file requester?
Amiga Dispatches

Ripple effects, why things are how they are and the HELP key

by Don Curtis
Copyright © 1989 Donald A. Curtis

Don Curtis is a police officer in Denver, Colorado. For the last couple of years, he has been assigned to program design and development along with system design and maintenance of 10 AT&T Unix 382 computers. In his spare time, Don is an assistant Sysop on CompuServe’s Amiga forums.

Time sure flies when you’re having fun; the last few weeks have been very hectic at work. We’re taking several major database applications from the city’s IBM mainframe and moving them onto our in-house UNIX minis. The only thing that was common was the data itself, and even that required extensive manipulation moving it from one system to the other.

All the programs had to be re-written from scratch, screens re-designed and concepts re-thought. The project has spanned several years, but most of it came together over the last few weeks and as with all new systems, problems developed.

When you deal with computers and computer programs, you must assume the end-user knows nothing about computers at all. You must also assume that if they can do something wrong, they will. The end user doesn’t have to know what’s going on ... they should only be concerned with the end product. Can the computer or program do what the users want it to do, in a timely fashion?

Not only that, do the users understand the consequences of changes they may request in the system? What are the ripple effects of a small change here and there? Let me give you an example:

In Denver, we have our main streets laid out mainly in a north-south/east-west grid. However, the downtown streets are laid out in a northwest-southeast/northeast-southwest grid. If that’s not enough, both the downtown streets and many of the east-west streets have the same name, but different suffixes. That is, downtown there’s a 16th Street, while out of the downtown area, there’s a 16th Avenue. Adding to that, if there’s no prefix to the street name, the defaults are north or east depending on which way the street runs, unless it’s a downtown street and they have no prefixes! There’s even some more things to add to the confusion, but there’s no need to go into all the gory details. Suffice it to say, that the address ‘510 16th’ is useless. It could be any one of 3 places in the City.

Our users wanted to be able to look up all incidents that may have occurred in a certain block in the city. That’s no problem. But one of the users wanted to make it easy on himself. He wanted to be able to look up, with a single query, all the incidents in the 500 block of 16th whether it was E. 16th Avenue, W. 16th Avenue or just plain 16th Street.

That sounds like a good idea - we often get information third or fourth hand that someone was seen on a street on a certain day and it might be useful to see if the police made contact with that person. The programmer (an outside consultant) also thought it was a good idea, so he put code into the system that inserted a wildcard character into the place the street suffix would go if the user didn’t enter a suffix. That way, if you just said ‘500 block of 16th’ rather than ‘500 block of W. 16th’, you would get all incidents on any of the three streets.

Great! But what no one thought of was the fact that using a wildcard in a query forbids the use of an index! That meant that if I entered a general query on the 500 block of 16th, the program would scan the entire database (several hundred thousand records) and look at the address to see if it matched the wildcard. And since downtown streets have no prefixes, you couldn’t look up just incidents on downtown streets, you had to look for them all since the wildcard was automatically inserted.

What was the fix? I removed the automatic wildcard and made the system look for exactly what the user entered. Now the system was able to use its index for the search, and queries went from 20 minutes to 10 seconds! You could look for all three locations by exact matches in less than a minute, far more efficient in the long run. I went back and explained this to the user, and he was perfectly happy.

This was a classic case of the programmer not understanding the user or the consequences of his actions. All the user really wanted was a quick way to look up generic addresses. The key word was ‘quick’, but the programmer instead thought of ‘easy’ and wrote the code forgetting, or ignoring, how it would effect other types of searches.

You can’t expect the users to understand how the program works, but you should take the time to sit down and explain to...
them why certain things work the way they do. You don’t have to get technical, a simple explanation is usually enough.

And the ripple effects? It took me hours to find the problem, but a single keystroke in an editor to fix it. The users went from being unhappy about the length of time certain things took, to being happy with the change.

Remember this little story the next time you wonder why things don’t seem to work the way you’d like them to work. Also remember it the next time you feel like flaming Commodore for not producing the feature you want in an Amiga.

If you write a program for yourself, you can customise it to your heart’s content. You don’t care if someone else likes your method since no one else will use it. However, add a second potential user and you’ve got a possible conflict between your way, and their way. The answer is a compromise. Obviously the more potential users of a computer or program, the greater the potential for conflict. You can give the users choices to help let them customise the program for their own purposes, but you can’t make it perfect for everyone. That’s just not possible.

The same is true for hardware, but to a lesser extent. The base hardware of the machine pretty much has to be the same for all users. That means compromises have to be made, and some of those compromises may not meet with your approval. Take the standard high resolution screen of the Amiga. It’s in interlaced mode to make it conform to TV standards. That way, you can take the output and minimally process it and put it on the air, or tape it or whatever. That’s good ... for folks interested in video applications. That’s not necessarily good for the rest of the Amiga community. As you know by now, using interlaced mode, with a short persistence monitor (as the vast majority of monitors are) causes screen flicker. Screen flicker can be annoying ... well, no, it is annoying.

When the Amiga was first brought out, it’s graphics capabilities rivaled those of any computer on the market for under $50,000 or so. Video tape recorders were now common home appliances. Desktop video was almost unheard of, but the potential was there. With the Amiga, a home user could suddenly produce ‘Saturday Morning’ quality cartoons at home. But for the home user to take advantage of this, the Amiga had to be able to be recorded. Thus, it needed interlace mode. The other option would have been an expensive video adapter, similar to those used on other computers.

With interlace, it was all there. The computer with it’s custom graphics chips, large number of colours, 640x400 resolution and the ability to directly tape the output made it a natural, so much so, that not only did the home users pick it up, but so did many professional video shops. If you think that wasn’t important, go out and try to find an Atari ST running a video application. If you find one, let me know where. I’d be interested in how they did it. What they had to add to the machine to make it work, and why they didn’t choose the Amiga in the first place.

Without interlace, the Amiga was still a wonderful computer, but it wouldn’t have fit all the niches it does now. I don’t believe that it would have sold as well as it has, and apparently either did it’s designers.

These are the consequences I’m speaking of. The initial buyers of the Amiga were hackers and folks interested in its video capabilities. It took a while for others to catch on and see the other features of the Amiga that made it stand apart from anything else on the market. If the initial buyers hadn’t been there, the Amiga may not have made it. Commodore wasn’t in very good financial shape back then and the Amiga’s success was a major reason that the financial problems are all in the past now.

There were lots of decisions like that made in the design of the Amiga. When you get upset with Commodore for something you don’t like, try to think of the consequences of having it some other way. When you put it all together, I think you’ll find that Commodore made some pretty smart decisions after all (and some fairly dumb ones also) when the total effect on the user community is considered.

Disk drive selects

That brings up another whole discussion. Why did Commodore make the A2000 external drive DPZ: and why did programmers hard code their programs to look for DPW?

On a disk drive, floppy or rigid, there’s a control line called Drive Select that’s used to make the circuitry active. Most computers have the ability to control several disk drives, and the Amiga is no exception. They do this by having the ability to apply a signal to one of many select lines thus signaling to the disk drive attached to that line ‘Get Busy!’

The general methodology is to use a jumper on the drive itself, or to have a twist in the drive control cable so the select lines attach differently. The result is that only one select line is actually hooked to the drive’s circuitry. Thus the drive only becomes active when that particular select line is chosen. If you didn’t do this, what you’d have is that when you attempted to access one drive, all the drives would turn on and nothing would get done - the system would become hopelessly confused.

To add a drive to a circuit like this requires the user to have knowledge of how to set the select jumper, or twist the cable. While moving a jumper is fairly simple, you usually pull a jumper off of one set of pins, and place it on an adjacent set of pins; many users are afraid to open up the drive and ‘play’ with them, so they have a service shop do the installation for them. That’s added expense, time and trouble.

The Amiga used a different philosophy. It was decided that they would have their circuitry set up so that you could take any drive, and put it at any position in the drive chain, and not have to make a jumper change. All the drives are jumpered the same. They are all jumpered as Select 0, the first drive in the circuit. Now the Amiga has the same multiple select lines
coming out of the controller chip (8520), labelled Select 0 to Select 3. The first line (Select 0) is always attached to the internal drive, thus it's drive DFi:, but Commodore performs a little magic on the other 3 select lines to allow the drives to all be the same.

If you'll look in your Amiga manual, you should find a diagram of the disk drive connector. You'll see 3 pins labelled Sel_1 to Sel_3. This is true for the connectors on the drives themselves as well. On the Amiga A1000 and A500, the external connector has all three lines hooked up. On the external disk drives, all three lines are hooked up on the input connector, but only two of the lines are hooked to the output connector!

On the drive, the line attached to the Sel_1 input connector is attached to the drive itself as the 'Get Busy!' line. The remaining two input lines are passed to the output connector, but shifted one position. That is, input line Sel_2 gets connected to output connect line Sel_1, Sel_3 goes to Sel_2 output and there is no connector on the output line Sel_3.

See the magic? If you have 3 drives on your system, drive DFi: should be activated by the computer's Select 2 line. When the computer activates Select 2, the signal gets passed out the back port into the first drive's Sel_2 input. From there, it simply gets hooked to the Sel_1 output and passed on to the second external drive. The second external drive gets it on the Sel_1 input, so it is connected to the drive itself and activates the drive.

That's similar to twisting the wires, but the system does it all for you, you don't have to worry about it. However, the A2000 is a slightly different animal. It can have a mix of drives inside it. There are several combinations possible, and one of those combinations is two internal floppies, DFi: and DFi:.

Since that's possible, they couldn't run the Select 1 (select line for DFi:) line outside the machine because that means it would possible to have two drives attached to the same select line, and nothing would get done because both drives would become active at once.

Because of the design of the external drives, it wouldn't work to have you move the jumpers around. The lines wouldn't be passed properly. So a decision had to be made. Either design the A2000 to use a completely different drive setup, or have the first external drive recognised as DFi:.

The consequences of the first choice would have meant that folks who moved up to the A2000 from the A1000 or A500 wouldn't be able to use any existing external drives they had. Dealer's would have had to stock two different types of external drive and the user's would have had to learn how to set drive select jumpers or pay a shop to do the job for them.

The second choice made much more sense. There's nothing magic about a drive addressed as DFi:, the only drive that counts is DFi:. You must have that to boot the machine (prior to autoboot that is). The users could keep their current external drives and just plug them in. The drive chaining logic would continue to work, and the 3.5" drive bay could still be used for either a hard drive or a second 3.5" floppy.

So what was the problem? Some software authors had hard coded their programs to look for disks in DFi: specifically! Those programs never looked beyond DFi: to see if there may have been a DFi:. Prior to the A2000, that was a safe assumption, but now the second drive can be either DFi: or DFi:. And whose fault was that? Was Commodore wrong for allowing a second drive but no DFi:, or were the software writers wrong for making assumptions about how the machine is configured?

Commodore has stated from the beginning not to make assumptions about the configuration of the machine beyond 256K and a single floppy labeled DFi:. While at this point it may seem safe to make the assumption of 512K, even that's not true. There are still 256K A1000s out there. That leaves the fault squarely at the feet of the software authors who made assumptions about the machine's configuration. And that comes right back to the idea of changes and ripple effects.

A simple change in design broke some programs because the authors chose not to follow the rules.

The HELP key

Larry Phillips is the author of the 'ViewPort' column, and another of the sysops on CompuServe. He and I have some severe differences on how the HELP key should function. We argued about it in the Amiga Tech Forum on CompuServe, but decided to bring it into the pages of this magazine and let you, the readers, decide how things should work. Not that any of this will be 'official' or carry any weight with Commodore, but it should be thought provoking ... and thus, useful.

A quick rundown of Larry's method of implementing the HELP function is that it should be context sensitive based on mouse position and what the program was doing at the time. Essentially, the way I understand his methodology is that there is a universal 'help.handler'. Through this facility, when the user presses the HELP key, based on the user’s preferences and the program’s implementation, the system looks at where the mouse currently is, and gives help on that specific object whenever possible. The forms of help given can be program generated, via a 'help' text file the user creates or a combination of both. One thing Larry definitely insists on is that when pressing the HELP key, the position of the mouse is paramount and without regard to whether the mouse is in the current window or not.

Generally, my idea is rather simple. First of all, the HELP key is just that, a key on the keyboard and like all keystrokes (except mouse shortcuts) should only go to the currently active window. Second, by its very nature, help is specific to a particular program. The help information for Program A is of use only with Program A. Because of that, it's up to the
program to offer help, not the system. Third, tying help to the mouse position can be a pain in the butt. I generally keep the mouse out of the way. If I hit the HELP key (like right now), I'd get help with the scroll bar, when all I wanted was the quick reference card to pop up. Under Larry's idea, at the least, I'd have to move the mouse out of the scroll bar into a 'generic' area to get my quick reference card. Finally, I don't like the idea of the manual being built into the program.

Most importantly, I feel the HELP key should go to the current window. Almost everything else you do on the Amiga with the mouse goes to the current window ... or makes the window current. If I want to close a window, I move to the close gadget and click on it. All that does is make that window active, and inform the program the user has selected the close window. It's up to the program owning that window to actually do the close. Larry's idea is similar in this respect, except he doesn't want the keypress of the HELP key to make the window active, just to notify the program that the user pressed the HELP key. If I'm working on Program A (it's the active window) but also have a window open for Program B, nothing I do with the mouse should have any effect on Program B except moving the mouse into Program B's window and clicking on it.

During our discussion on CompuServe, Larry felt that the HELP key should be treated differently to the other keys. He felt that there were current keyboard shortcuts that were the equivalent of the mouse keys, so the HELP key wouldn't be any different than those. My objection to that point is that there is no equivalent to a mouse HELP key. Thus tying the HELP key to the mouse makes it a shortcut for a non-existent mouse key or a keyboard key that is intimately tied to the mouse and mouse only. In other words, I feel it would be a complete oddity. By tying the HELP key to the current window, it then becomes a "normal" key, handled just like any other keystroke.

Look at it this way: both Larry and I are working on Program A. We both decide we want help on Program B. Under both of our systems, we must first move the mouse into Program B's window. Larry would then have you simply hit the HELP key. I would have you first click on the left mouse button and then hit the HELP key. Our hands were both already on the mouse so it's no extra effort to stop typing and reach for the mouse, all that's needed is to move one finger, nothing else. At this point we're arguing over a simple mouse click. Silly? I don't think so.

The implications in that mouse click are huge. First, I've told Program A to quit accepting input and told Program B to start paying attention to me. Second, I've now allowed Program B to devote it's full attention to my needs. Since any help on Program B must naturally be generated by Program B, there is no ambiguity about which program is going to offer assistance, B is active, B will help me. What if Program A and Program B are both word processors? What if I'm comparing the two and since I'm not familiar with either of them, I need assistance? And what if I like to keep the mouse out of the text area, so move the mouse out of my current window and it happens to be lying on Program B's window?

I hit the HELP key and under Larry's setup I'd get a quick reference display for Program B. The quick reference card tells me that Alt-E is the shortcut for the menu item 'Edit new file'. I hit Alt-E and Program A's text suddenly gets Erased because Program A's (remember, it's still the active window) interpretation of Alt-E is "Erase Window". Of course, any program should at least warn you before erasing the entire window, but that's not the point. The point is that it's too easy to get confused.

How's this for an alternative approach? If help only goes to the active window and help is only under program control, not system control, then when I hit the HELP key, I get a requester asking what I want help with. There could be buttons for 'Generic Help', 'Help with Gadgets' or 'Help with Menus' (plus whatever else the author decided to implement). I select 'Help with Menus' and get a requester telling me to select the menu item where I need assistance. Since all gadgets, menu items and so on are under program control, it's no problem for the program to offer help with 'Quit' rather than quit the program. I simply select the menu item just as I would when running the program. The program simply notes I'm asking for help, not asking for the selected action to take place and gives me assistance. All with one hand too, no awkward holding of mouse and help key at the same time.

Of course, both implementations would require the program to offer that level of help. I disagree with that much code and program space being used to repeat what's in the manual. I much prefer that the program-generated help be more of a quick reference card. I hit the HELP key, and a quick reference requester pops up giving the most commonly used keystrokes and listing the manual page for additional assistance. I see no reason to help the pirates by providing an on-line manual with the program. Removing on-line manuals won't stop software theft, but it also won't increase it. If Billy Bluebeard can simply copy a program and not worry about copying the 300 page manual he's more likely to simply make a copy than he would be if he also had to make a copy of the 300 page manual.

I'm not saying penalise the honest user. Most people forget the keystrokes for a specific action, and just need a quick reminder. Just hit the HELP key, then a RETURN (to close the help requester) and keep on going. Easy, simple, painless and a snap to include in a program. Including help with everything in the program (Larry had no objection to even 'help with the scroll bars' being in a program) would require not only room in the program code for the text, but also logic in the program to decide the context and specifics of what help is actually needed.

Well, that's my ideas, and objections to Larry's ideas in a nutshell. It's up to you, the user and developer to decide how you want things implemented. I'd be interested in your thoughts on the subject. Larry and I are just two Amiga users, it's not important what we think as individuals. What we're really interested in is what the Amiga community thinks as a whole.
Access

The world of non-commercial software

by Steve Ahlstrom
Copyright © 1989 Steve Ahlstrom

As the primary Sysop on CompuServe's AmigaForum, and a veteran Amiga programmer, Steve Ahlstrom sees just about every notable public domain and shareware program that gets written. In this column, he tells you about the ones that have recently caught his eye. If you have information you would like to share through this column, you can leave a message for Steve on AmigaForum (76703,32006) or send it to us, and we will pass it on.

There seems to be quite a bit of confusion concerning shareware, freely distributable copyrighted software, and public domain software. The confusion seems to be among the software authors as well as the users.

Shareware

A shareware product is actually a commercial product with a non-standard distribution channel. The author asks for compensation for his work, usually in the form of currency. The distribution of the program may take many forms. The author may release a fully functional program. The documentation will state you may 'test drive' the software. If you find the software meets your needs you are to mail a check for the purchase price to the author. An example of this type of program is Access! by Keith Young.

Another common method used to encourage payment for the software, is for the author to release a semi-functional demo of his program. In this case, not all features of the program are functional in the freely distributable version. If you want the full version you must send payment to the author. Rick Stile's uEdit falls into this category.

The above distribution methods are the most common. There are some twists. In return for your payment you may receive source code to the program, you may be put on a mailing list to be notified of future updates, you may even be mailed updates as they become available. Some shareware authors may request non-traditional payment; payment should be made to non-profit organisations, payment made in goods (such as diskettes) rather than currency, and various other idiosyncrasies.

Regardless of the distribution or payment channels, shareware programs are not free. They are copyrighted works and must be paid for. As with more traditional software publishers, support given by the shareware publishers can range from excellent to poor. The advantage to shareware products is being allowed to use the program extensively to determine its suitability to your needs.

Freely Distributable Copyrighted Software

This class of software for the Amiga seems to be growing. This is software the author wishes to make available to anyone who wants to use it. No compensation is required. The author does claim ownership to the software and has the right to dictate how the software will be distributed.

There are many reasons why an author may chose this route. Quite often the program may be the basis of a future commercial product. The author may want to share concepts but not give away the farm. This goes against the grain of the 'hacker ethic' which says that all knowledge and source code should be made freely available to anyone who wishes it.

This concept is generally acceptable when a computer system is new. Everyone is trying their best to learn the intricacies of the system make their knowledge available to everyone else. As the machine matures there are more people programming the machine and techniques derived may become commercially significant. By not retaining control of the program the author may find himself, at some later date, competing with another product using his own code.

Many times this type of work is called FreeWare.

Public Domain

There are no restrictions on public domain software. Once someone has declared that their work is public domain no-one can claim any rights of any type on the software. Anyone may use it for any purpose. It may be included in the code in other works without permission (because no one exists who has the right to grant or deny permission) or it may be sold as a product with no compensation to the original author.
Here is where the confusion lies. Many times I’ll see a documentation file for a program that specifically states 'This is a public domain program'. Then, later in the documentation, the author outlines the method of distribution. In these cases the author is being contradictory.

By declaring his work to be public domain he has given up all rights to it, including the method of distribution. He no longer has any control over the work. If the author wishes to retain some control over the work he should attach a copyright notice (and register the work with the copyright office).

Shareware Ethics

If you wanted to sell an item, say a lawn mower, and your neighbour said he was interested but wanted to use it first to see if it was all you said it was, you would probably lead it to him. You fully expect your neighbour to either pay you for the lawn mower or return it.

This is much the same with a shareware program. The author allows you to give his product a fair evaluation. If you use the program, the author has every right to expect payment.

While a lawn mower cannot be copied, computer software can. In most cases the price of the product is much lower than a more conventionally marketed comparable product. There are no middlemen involved to increase the cost of the product. Traditionally, a distributor will take 10-15% of the suggested retail price and the retailer will get 10-40%, depending whether or not the software is sold at a discount. A shareware product can be priced at 55% less than a conventionally marketed product. In fact, shareware products are normally priced much lower. There is little or no distribution or packaging expenses, and, for those products distributed via networks and BBS, there is no media costs.

Shareware, if it meets your needs, is an incredible bargain. Pay for it. Most people would never consider shoplifting a product from a computer store; don't continue to use unpurchased shareware products.

Key Map Editor

Freely distributable Software © by Tim Friest

One of the Amiga’s strongest points is that it really is the computer for the rest of us (in total disregard of Macintosh’s claims). Amiga allows easy configuration in just about any area. By selecting your printer from Preferences you have configured your computer to work with practically all off the shelf commercial products. You may change colours, screen resolution, font sizes, mouse and key speeds and make these the defaults. Unlike other machines whose ambience implies 'do it your way or don't do it at all', Amiga puts you in control of your computing environment.

Now, thanks to Tim Friest’s Key Map Editor, there is an additional tool to customise your computing environment.

What are keymaps? They are the files hiding in the dev$:keys directory labeled usal, gb, is, dk, and so on. Amiga is an international machine. According to Commodore, 70% of the Amiga’s worldwide sales are outside of North America. Keymaps allow the machine easy penetration into worldwide markets.

With keymaps, machines do not have to be custom manufactured for each country in which it’s sold due to language or alphabet differences. All that is necessary is that a keymap for that country be written and then loaded from the startup-sequence. For instance, in the usal keymap, there is no British monetary symbol. Load the British keymap (setmap gb) and shift-3 becomes £, the British pound symbol, replacing the # in the usal keymap.

Keymaps do make it easier for the internationalisation of the machine. That is not their only use, however.

When the typewriter was invented, some hundred odd years ago, various keyboard layouts were tried. After much trial and error, disagreement, and confusion, the QWERTY system, that most of us use, became the standard. It is not the most efficient for typists. Different layouts can dramatically improve speed.

Why did it become the standard? In those days, the limiting factor was the mechanics of the typewriter. The physical mechanism that produced a character on a piece of paper from a pressed key was the bottleneck. Type too fast and the typewriter would easily jam. QWERTY was devised and became the standard because it was inefficient and slowed typists down.

Now, of course, there is no such limitation. Various alternative keymaps have been devised that are much more efficient for the typist. The most popular of these is the Dvorak keymap. Once learned, it's use will greatly improve your typing speed. Just try to find a Dvorak typewriter, though. Or try to find any other computer than will allow you to change its keymap to Dvorak and have it work with practically all programs available for it. With Amiga, it's a cinch - setmap usa2.

Hopefully you will agree what great and marvelous things keymaps are and how the ability to change those keymaps make the Amiga stronger and more well rounded than other machines in its class. You might ask, though, why should I be excited about a Key Map Editor? Since the computer comes with a variety of keymaps and CBM is surely going to create keymaps for any new country the machine is marketed in, what use is such an editor to me? Because the pressing of 1 key does not mean that just 1 character will appear on the screen.

With the Key Map Editor, you can transform the keys you seldom use (or SHIFTed, ALTed, CTRLed variations of those keys) into sending practically any textual information you want to your application. Key Map Editor is a very powerful keyboard macro creator.
As a very limited example, I have redefined my numeric keypad. It now looks like the following:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>/</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>run dmam&lt;cr&gt;</td>
<td>8</td>
<td>9</td>
<td>-</td>
</tr>
<tr>
<td>acc&lt;cr&gt;</td>
<td>whap&lt;cr&gt;</td>
<td>6</td>
<td>+</td>
</tr>
<tr>
<td>direct&lt;cr&gt;</td>
<td>2</td>
<td>3</td>
<td>enter</td>
</tr>
</tbody>
</table>

So far, I've only redefined five keys and those only in their non-qualified state (that is non-SHIFTed, ALTed or CTRLed).

Instead of displaying a '7' when I press the 7 key on the numeric pad, my keymap sends a `RUN DMAN<CR>` to the active window. DMAn is a directory utility that I use. Eight keystrokes are saved in running that program.

You can get a bit more complex. Instead of displaying a '5' when the 5 key on the numeric pad is pressed, my keymap sends `WHAP<CR>`. WHAP is an ARexx script (I use William Hawes WShell so no preceding RX is necessary). WHAP could just as easily be an AmigaDOS execute script. Just one keystrokes causes an entire chain of events to occur.

Key Map Editor is very handy! Should you wish to create keymaps for various applications, you could have a key on your default keymap `setmap` another keymap then run your application.

There are a lot of possibilities here. Because of the combinations of the various qualifying keys, each key on the keyboard can respond in up to eight different ways. This gives you hundreds of combinations for user defined macros. Any program which is correctly written and uses the default keymap (rather than a keymap internal to the program) will fully support your modified keymaps.

Key Map Editor is a complex, yet easy to use program. Everything you need to know is in its documentation.

**BindNames V1.0**

*Public Domain software by Dave Haynie*

If you have a hard disk, do you find it inconvenient to be constantly editing your startup-sequence file for logical assignments most new programs require? You add the assignment, reboot, then find out that your assignment is dependent upon another assignment that had not been made yet?

Dave Haynie comes to the rescue with **BindNames**. BindNames is designed to do all of the logical name assignments at once. You create a subdirectory in your root directory called *Names*. In this directory you create a file which contains your assignments. For your system assignments, you might have a file in `SYS:Names` called `SYSTEM` that looks something like:

```
BIN: SYS:bin
OS: SYS:os
C: BIN:c
COM: BIN:Com
L: OS:L
FD: OS:Foms
S: OS:s
DEVS: OS:Devs
LIBS: OS:Libs
ENV: RAM:Env
T: RAM:T
```

From your startup-sequence you would replace all the assigns with a call to **BindNames**:

```
BINDNAMES SYSTEM
```

BindNames will read all name files before making any assignments. It will figure out dependencies. It does not matter how you order the names. It will create directories it cannot find, such as the `RAM:Env` and `RAM:T` in the above example, and will generate warnings for assignments it cannot resolve.

Instead of 'assigning the world' when you power on the computer, you can make some specific assignments only when you need them. If you want to run a program which requires logical assignments, just create a file to put in the `SYS:Names` directory listing the needed assigns. Before you run that program you would type `BINDNAMES <filename>`. You could go further and create an ARexx or AmigaDOS execute script that will do it all for you - then add the command to your custom keymap!

**KeyHack**

*Freely Distributable Software © by Jes San*

In Volume 2, Issue 1 of TransAmi, I talked about a program called **SetCPU** 1.4 by Dave Haynie. SetCPU allows to copy the system ROM into 32 bit RAM (if you are using the A2620 board). This greatly increases the speed of your system.

Unfortunately, the early keyboards that were shipped with the A2000 are not compatible with SetCPU. These keyboards are relatively easy to spot. They are made by Cherry and have small sized function keys across the top. If you own one of these keyboards, and an A2620, you may, with the use of Mr. San's KeyHack, use the SetCPU program.

KeyHack is a quick, temporary hack, to patch a RAM copy of the Amiga OS to slow down keyboard handshake.
MemWatch
Freely Distributable Software by John Toebes
© The Software Distillery

MemWatch has been around a while, but many times we tend to overlook the older programs for the flashy newer ones. Newer does not always mean better. In the case of MemWatch, it is a classic that has an important use for your system.

MemWatch is a relatively simple program that can help prevent random gurus. Run it from your startup-sequence. MemWatch will sit in your system quietly, watching. It looks for errant programs that change the values in the first 100 bytes of memory. Few programs write to low memory intentionally. This is usually a side effect of referencing a NULL pointer. In many cases, writes to low memory are not harmful. Other times they are timebombs just waiting to welcome the turbaned one. It all depends on just what addresses were overwritten.

When run, MemWatch saves a copy of the first 100 bytes of memory. When a write to low memory is detected, a comparison is made with its copy of the memory values. Should the values be different MemWatch restores the correct value and then puts up an alert to let you know that low memory was overwritten. The alert contains the value that was written and the address to which it was written.

To ensure that MemWatch does its job it must check low memory frequently so any damage can be repaired before it propagates. By default, the program checks low memory every 5000 microseconds. This timeframe is user configurable.

There is no guarantee that MemWatch will catch and repair all damage to low memory but it will most of the time. It is a very useful program to have running the background when you are developing your own code; it is an extra debugging tool.

It also is useful in finding those programs that are not written correctly that do damage to low memory. Some of them, like Aterm 7.3, write to low memory but do no apparent damage. In the case of Aterm 7.3, it writes an 0x04 to location $00. Other programs may do more damage and eventually lead to unexplained system crashes. MemWatch will let you know which programs you might have to watch more closely.

Guru
Freely Distributable Software by Mike Haas
Freely Distributable Software by George Musser, Jr.

'Guru meditation' is a very familiar phrase for all of us. For most of us it just means one thing, the system has crashed. We are given cryptic numbers that explains why, but, in and of themselves, the numbers are mostly meaningless. There are few sources readily available that contain a list of the error numbers. One source is the from the excalaires header file from your C compiler, assembler, or from the Native Developers Disk from Commodore. Another listing is contained in the COMF manual. Many people may not have any of those sources available.

There are a couple of programs available that allow you to type in the error portion of the guru number (the number to the left of the period is the error, the number to the right is the address at which the error occurred) and give you and explanation of what the error means. The explanation given may not be all that meaningful but much more so than just a series of numbers.

Both of these programs are named GURU. Each perform similar functions. In most cases the program by Mr. Haas has better formatted and more informative output. C source for Guru by Mr. Musser is available.

LED
Freely Distributable Software © by Khalid Aldoseri

I was talking to Khalid in AmigaForums' conferencing area. I mentioned it would be nice if there was a way to tell when activity was going on in RAM:. Using ARC or ZOO. I almost always decompress files to RAM:. Sometimes large files can take a bit of time while decompressing. I get impatient then I start to wonder if the machine has locked up. Khalid said 'hold on a sec!' and uploaded LED.

LED will flicker the Amiga's power LED as data is being read or written, to or from, any specified device. RAM: is not a device, so it does not work there, but ASDG's VDO: qualifies.

The program is a little disconcerting to use. We've all trained ourselves to expect to meet the GURU when we see the LED flash. After using LED for a while I still find, myself reacting negatively when I see the LED flashing from the corner of my eye.

But, the program works as advertised. I no longer wonder if ARC or ZOO is still 'doing it's thing' or if I've locked up. Now I wonder if LED is running properly if I'm about to meet the GURU!

LV
Freely Distributable Software © by Khalid Aldoseri

Khalid explained that while he was playing with flashing the power LED he thought of another use for it. LV was born.

Have you rebooted your system only to find that your floppy or hard disk was not validated? You were probably too quick on the three fingered salute.

This used to be a major problem with floppies under 1.1 of the operating system. A write would be done, the LED on the floppy would turn off, and update would be done, and the LED would come back on. If you rebooted before the update you'd invalidate the floppy. The Amiga took a lot of abuse from its detractors - it crashed all the time and trashed floppies.
Since 1.2 the floppy LED stays on until after the update is completed, but you can still be a bit too quick on warm boot. LV helps. Every time a write is done to a device Amiga's power LED will dim. After the update is done it will return to normal. What this means is if the LED is dim, don't reboot.

Khalid Aldoseri is on CompuServe and can be contacted by sending email to 75166,2531.

FixIntuition  
by 'Roger'  

Have you run across the Intuition bug that causes GURUS when you switch screens? This usually occurs when you have two interlaced screens and one non-interlaced screen.

I usually get hit by this at least once a day. This is a known bug and hopefully will be fixed in 1.4. Until then, there is a very small program to fix the problem called FixIntuition. Just run it from your startup-sequence and these lock ups will go away! I'm not sure who the author is. The short readme file is just signed 'Roger'. Well, Roger, thanks! You've solved a very annoying problem.

Availability  

For those registered on CompuServe, all the above items are available in the CompuServe AmigaTech libraries. [You will also find some of them on this issue's TransAmi disk for convenience - ED].

Program news  

There have been a lot of reports of an Access 1.43. There is no such beast; at least, not yet. Keith Young has been spending time working on a couple of commercial projects: He was working on Access! in his spare time. Unfortunately he experienced a hard disk crash that was totally unrecoverable and some code was lost. The good news is that he expects to have a new version in the near future. Keith does not want to commit to exactly what new features the next version will have but full ARexx support is a definite.

Khalid Aldoseri is working on a program called Scripl.. This program is a real winner. Khalid has turned down at least two offers for it to be a mainline commercial product. He intends for it to be freely distributable. It is a scripting language that allows you complete control over any AMIGA program. ARexx is not required nor is anything else other than the program itself. Look for a full review in a future column.

STOP PRESS  

Khalid's Scripl will be on this issue's TransAmi disk!

Also, look out for articles by Khalid in future issues of TransAmi.
And Now ... 1.4!

A look at what might be in 1.4 ... and then again, might not!

by Mike Todd

With the advent of the Enhanced Chips Set (ECS), and with a market increasing its demand for a more polished front-end, the development of a new release of the operating system has been a top priority in Commodore.

At the recent Developers' Conference at Nottingham University (see the report on page 29), Carolyn Scheppner and Gail Wellington of Commodore US, gave an insight into what we can expect in this new release.

Unlike version 1.3, which was really just an incremental revision of 1.2, 1.4 is a major re-write of the operating system aimed principally at supporting the RCS. However, the opportunity is being taken to refine existing features and to add others that developers and users have been screaming for.

Developing such a major release is no easy task and the whole process starts with a very long wish-list. On it are all the features which the engineers, marketing boys and yes, even end-users want to see in the new release. Development then continues apace, with much effort being concentrated on the essentials, with an Alpha version being provided to key developers and in-house testers - then, at some point someone has to say 'STOP! this is the final feature list' at which point the release moves into the Beta-test phase.

Many more developers will then receive Beta-test versions, which are re-released as bugs are reported back and knocked out of the system. Once we see a release version on the horizon, development moves into the Gamma stage (or at least 1.3 did!) and then on to final release.

Well, 1.4 is still only in its Alpha-test stage and the final feature list has still to be decided. The only thing that we can say with any certainty is that it will have the support for the ECS, since that is the main motivation for its development.

Another feature which seems almost certain (but can't be guaranteed until the final ROMs are produced) is that it will move up to 512k of code. Memory space from $80000 upwards has always been reserved for the operating system, although only the upper 256k has ever been used to date. A1000 owners may have a problem with this, since their machines have the operating system soft-loaded in a limited 256k of special RAM - will Commodore provide a 1.4 upgrade mechanism for A1000 owners (it wouldn't be difficult!), or will we see the A1000 users being left behind? Only Commodore knows the answer.

The following is a feature and wish-list which Commodore expects to develop into new releases of the operating system in the future. Although the list assumes that the features will appear in 1.4, it cannot be stressed enough that there is no guarantee at this stage.

With the transition from the Alpha stage to the Beta stage due any time now, the big decisions about what will and what will not make it into the new release have got to be made. Clearly, there are some features that must go in - support for the ECS for instance; however, other really neat features will not go in unless they can be guaranteed as near bug-free as possible.

Many of the following features will come with 1.4, others with 1.5 and others may well fall by the wayside.

- ARexx will be supplied as part of the operating system, making the Amiga the first microcomputer to have this inter-process communication as standard
- Console device character maps will be provided, which should make text cut and paste within a console easy
- It will be possible to notify a program that another task has just changed a file
- There will be record locking, to prevent other tasks (and other users on a network) writing to a record while it is being updated
- Font scaling is being developed - this will allow the right-sized font to be selected - rather than put up with a compromise of the nearest the system can find
- Overscan will be fully supported through preferences
- Text display will be much faster
- The fast file system will be in ROM which will make it available to floppy users
It will be possible to auto-boot from any device, which at last means booting from floppy drives other than D0:

New single precision IEEE math is to be supplied - giving a significant speed increase, but at the expense of accuracy

Workbench icon selection will be improved - there will be faster display of icons, drawer opening and so on

It will be possible to select an icon before all the other icons have been displayed - an end to one of the great frustrations of the WorkBench

The Guru is dead! There will be no more GURU meditations! Instead, there will be sensible (?) DOS/system messages

An optional full screen editor will be part of the system

The user will be able to turn off the irritating drive click

An auto-start-up drawer will be provided into which WorkBench users can put programs to be executed at start-up

There will be improved methods of icon selection, including a select-all facility and the ability to drag a box around the icons to be selected

An attempt is to be made to allow the Workbench user to be able to do as much as the CLI user - including giving access to files without .info files

The CleanUp WorkBench menu option will at long last take account of the size of the text under the icon

Standardisation is uppermost in the minds of Commodore, and system-standard file and font requesters (along the lines of those promoted by ARP) will be included

System message will no longer be in ROM, to allow alternative languages to be used

There will be a large, virtual-screen capability with mouse-controlled scrolling

The order of Exec's system start-up will be changed. Expansion will be set up earlier,cold capture will occur later and the exception vectors will be set later

Some changes to Graphics include a change to the copper lists and an increase in the size of the ColourMap

Changes to Intuition include support for overscan. The top and left edges of the screen may be non-zero or even negative, the left-Amiga select will be 'stolen' by Intuition and there will be some subtle changes to gadgets

The system-configuration file will go, although there is no indication of what will replace it

The serial.device will be modified to allow unit numbers to be used for multiple serial ports

WorkBench may slip out of the ROM and become a disk-based library

As I said earlier, this is a list of some of the features that users and developers have asked for, and which are under development at Commodore. However, it cannot be stressed enough that these features are not guaranteed to appear in 1.4 - or in 1.5 - or even in 1.6 - or ever! Whatever happens, it is Commodore's declared intention to improve the machine's image by making sensible changes to the system.

While Carolyn explained many of these features, she stressed the need for programmers to obey the rules; something we at TransAmi have been stressing for some time.

Version 1.4 will bear little relationship to earlier releases - everything will have moved, so woe-betide anyone who has jumped into ROM code, made assumptions about things in Exec, or used system-private structures or libraries or simply not obeyed the rules.

However, Commodore has made sure that anyone obeying the rules will have a smooth transition to 1.4.

All user manuals are to be re-written, and each machine will come with three basic documents: a machine-specific introduction and installation guide, a WorkBench guide and a BASIC manual (yes, Gail informs us that there will be a new version of BASIC supplied with the machine). For the advanced machines, it is hoped that these will ship with additional documentation giving technical information on the CLI, shell and startup-sequences for power users.

The major technical reference material is still in the process of being upgraded for 1.3 and supplements for 1.4 will be published.

Up to now, we have seen five versions of the operating system:

<table>
<thead>
<tr>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.0</td>
<td>Kickstart v1.0 now obsolete, but the original!</td>
</tr>
<tr>
<td>3.1</td>
<td>Kickstart v1.1 the first upgrade, for NTSC only</td>
</tr>
<tr>
<td>3.2</td>
<td>Kickstart v1.1 as 3.1 but for PAL users only</td>
</tr>
<tr>
<td>3.3</td>
<td>Kickstart v1.2 the first really solid release</td>
</tr>
<tr>
<td>3.4</td>
<td>Kickstart v1.3 as v1.2, but with autostart added</td>
</tr>
</tbody>
</table>

Version 1.4 will be the first major revision of the operating system and represents a major re-write in many areas. All developers and users can look forward to a significant increase in the performance of their machine ... but, don't hold your breath!

Although Commodore has already hinted at a release date by the end of 1989, most realists believe that Beta-test versions will not be released to developers until late in 1989 and the final release may not be until mid-1990. But it will be worth waiting for!
Assembly Language

Part 3 - reading a file, and some pure thoughts

by Jim Butterfield
Copyright © 1989 Jim Butterfield

Jim Butterfield needs very little introduction - his name is a household word among Commodore enthusiasts the world over. His association with microcomputers goes back to the 1K KIM-1. Jim's encyclopedic knowledge of Commodore's products is witnessed by his articles, books, lectures and even television programmes.

'Dear Jim,'

'In part 1, you implied that MOVE from a data register to an address register could be followed by a branch which would be dependent on the data moved - 'Tain't so.'

'Dear Reader,'

'Correct. Any command referencing an address register as its destination will not affect the flags which are tested by the Branch instructions.

'As it happens, the branch you are referring to (the one following the DOS library opening) will work correctly on current systems. The flag happened to be correctly set by the OpenLibrary subroutine itself. Try spelling 'dos.library' wrongly, and you'll see the branch at work, skipping the rest of the program logic.

'However, that's not rigorously correct coding. I'll try to watch it in the future.'

Files: Preliminary

We'll be reading a file this time, and putting its contents to the screen. Before we do, however, we should look into the concept of working memory. Our earlier program examples avoided this question; it's time to dig in there.

If you have programmed 'simpler' computers, you may have used the simple method of defining the work space you want, and then using it. That still works on the Amiga, but there are several other options that might do the job better.

We'll need to dig into programming methods, but also into the architecture of Amiga programs. It's a worthwhile trip.

Program Structure and Hunks

On simple computers, a program comes from storage into memory. Its location is usually known in advance, but the working rule is: the whole program comes in to one place.

The Amiga could not have effective multi-tasking if each program insisted on loading to a particular address in memory. There would be no way to resolve the conflicts between demands of various programs. So, as you undoubtedly already know, Amiga programs are able to relocate as they load.

But in the Amiga, programs don't need to load as a single chunk. A single program on disk can be made up of several hunks. As the program loads, the hunks split apart; each one finds an independent piece of memory to occupy. The Amiga loader makes sure that all the pieces connect together.

Why hunks, as opposed to a single program unit in memory? There are several reasons. First, it saves the Amiga the bother of finding a piece of memory big enough to hold the whole program; bits and pieces of memory can be found wherever they are. Secondly, certain kinds of data might call for particular kinds of memory; for example, a video screen within your program might need to go into chip memory, even though the rest of the program can go anywhere convenient. Perhaps most importantly, the hunk system lays the groundwork for future Amiga architectural development.

How do hunks happen? You can force your assembler to break your code into hunks, if you wish. You would normally do this if you wished to develop and assemble your code in separate pieces, and have the linker put them together. Even then, you can ask most linkers to coalesce several small hunks together into a single hunk.

Hunk history, Hunk types

For reasons that seem mostly historical, there are three types of hunk: code, data, and bss. This last stands for the near-meaningless 'Base of Stack Segment'; many years ago, I worked on a system which used a similar bss designation to mean 'Block Storage Size'.
The idea was this: program instructions go into the code hunk; predefined ‘fixed’ data go into the data hunk; and although the bss hunk contained no values (only a size definition), it would set aside memory for your working values. To follow the idea a little further: your program was not expected to modify the contents of the code or data hunks; all working values went to the bss area. You could disassemble a code hunk and find nothing but program instructions there; you could perform a hexadecimal dump of a data hunk and see only data, not code.

These rules were never precisely observed. Some compilers dutifully produce the three type of hunk, following which the linker might coalesce some of them or even discard empty hunks.

Data doesn’t need to go into data hunks. Putting data into code hunks especially strings such as ‘dos.library’ allows the information to be accessed by means of (PC) addressing, with corresponding savings in memory and time. Working memory, the equivalent of bss areas, could be used within any type of hunk. But as we’ll see, there’s a tendency for working memory to come from completely outside the program itself.

The usage rules for the three types has almost disappeared. For small programs in particular, everything could go into a single code hunk: program, data, and work space. Our example here will do that .. but we’ll use it to study the work space question more closely.

Work space and purity

Most programs need a place to store data. This might contain such things as variables, pointers, or data buffers. There are at least four ways we can get that work space:

1 Use a bss hunk to provide the memory space;

2 Define space within our program, perhaps in a code or data section;

3 Ask the system stack to give us space (using LINK and UNLK);

4 Ask the system to allocate a block of memory (using AllocMem and FreeMem).

With the first two methods, the space is within our program. The last two involve external space: we ask for it, and are granted the memory workspace.

Does it matter which style we use? Yes, if we want the code to be pure. Pure code will be able to run more than one job at a time; if we wish to make this program resident, the coding must be pure.

Here’s the idea: we’re about to write a program that will dump a file to the screen. To do this, the program will need to bring data from the file into a buffer. Taking the idea further: suppose we wanted the same program to handle two files at once. With a single buffer, we’d have a mess: one file would overwrite and destroy the buffer contents, and the work would get badly mixed up.

Options 3 and 4 above, however, allow the program to ask for two buffers. That way, the two files could be kept separate.

Program task

Our program will be called DUMP. When the user commands, from CLI DUMP FILENAME, the contents of that file will be printed on the screen. Unprintable characters will be changed to dots, except for NewLine, which will behave in the usual way.

One more thing: since the file might be very large and dull, we want to be able to stop the file dump. If we detect a CTRL-C key combination the job will terminate.

The first version of this program will not be pure. Let’s just get the job done.

Startup code

Let’s insert comment lines with name and date so that we can remember something about this program when we look at it in the future. Our data will be sparse here to save magazine space. The identity line is followed by system XREF definitions; you’ve seen most of those before.

; Program Dump - Jim Butterfield. March 15/89.
; Exec library calls
xref _LV0OpenLibrary $-228
xref _LV0CloseLibrary $-19E
xref _LV0SetSignal $-132

; DOS library calls
xref _LV0Open $-51E
xref _LV0Close $-24
xref _LV0Output $-3C
xref _LV0Write $-30
xref _LV0Read $-32A

Keep in mind that we have a ‘command tail’, whose length is in register D0 and address is in A0. We must save this information ... for the moment, we’ll copy it to A4 and D4.

;-- Initial setup:
Startup
move.l a0,a4 ; Remember ptr to argument line
move.l d0,d4 ; Remember len of argument line

Assemblers do a lot of ‘thinking’ for us. The first MOVE command transfers information to an address register. Formally speaking, it should be a MOVEA command. But the assembler figures that out and makes the adjustment.
Now, we'll open the DOS library in the usual way.

```
lea dosname(pc),a1 ; name 'dos.library'.
clr d0 ; Any version (0)
move.1 $4,a6 ; using Exec library,
jsr _LVOOpenLibrary(a6) ; Open DOS.
move.1 d0,a6 ; Remember Dos pointer.
beq.s StartupQuit ; Exit if error
```

If we did not take the above branch, we have opened DOS correctly and can now go to the next step. We'll call this as a subroutine, using BSR (branch subroutine). We know that the routine will be close by, so we'll tell the assembler that we expect a short (s) branch.

```
bsr.s DOSinit
```

When we come back from that subroutine, the job is done. The program will close the DOS library and terminate.

```
move.1 a6,a1 ; DosBase
move.1 $4,a6 ; Use Exec,
jsr _LVOCloseLibrary(a6) ; close DOS.
```

Here's our first level subroutine. It's not our main job yet. We'll obtain our output handle that will allow us to write our results. Then we'll put a binary zero character at the end of the 'command tail', which probably has a NewLine character in that position. We need that binary zero to create a C-language type of string; that's the kind that DOS will want for the file name we give it.

```
;-- Get CLI outhandle:
DOSinit
jsr _LVOOutput(a6) ; get CLI outhandle,
move.1 d0,a5 ; & then remember it.
;
;-- Change arg file name to 'C string' :
checklen
move.b #0,-S1(a4,d4.w) ; binary zero at end
```

The above is quite a combination of address elements. Start with A4, the address of the command tail. Now add D4, the length of the command tail we can use, w word length, since we may be sure that the command will not contain thousands of characters! The sum of these two registers will take us one beyond the position we want to go, so we back up with a displacement of -1.

```
move.1 a6,a1 ; DosBase
move.1 $4,a6 ; Use Exec,
jsr _LVOCloseLibrary(a6) ; close DOS.
StartupQuit
```

There should not normally be any spaces at the start of our command tail, but we'll scan to remove them just in case. We use the 'indirect with post-increment' addressing mode. Indirect means that we use the contents of A4 as an address; post-increment means that we add one to the address after we have used it. When we're finished, we back up one position.

```
moveq 40,47 ; zero column count
```

Here's where we read a block of data from the file. The Read command is defined as follows:

```
actualLength = Read(file,buffer,length)
```

```bash
move.1 a4,d1 ; filename pointer
move.1 #1005,d2 ; MODE_OLDFILE (for reading)
jsr _LVOOpen(a6)
move.1 d0,d6 ; file inhandle
beq.s DOSquit ; no good, quit
```

If we do not take the above branch, the file is open and we're in business. Now we will call our main job with another BSR (branch subroutine) call.

```
bsr.s DumpFile ; the main job
```

When we return, we must tidy up. That means that we must close the file. Close requires only the file handle in register D6.

```
move.1 d6,d1 ; use the handle..
jsr _LVOClose(a6) ; to close the file
DOSquit
```

We're finally on our way. DOS is active (A6 contains the library address), the file is open (its handle is in D6), we have an output path (handle in A5) and we'll be ready to read data after a little book-keeping.

```
; DOS is open and we have our in/out handles ;
DumpFile
moveq $0,d7 ; zero column count
```

Here's where we read a block of data from the file. The Read command is defined as follows:

```
actualLength = Read(file,buffer,length)
D0 D1 D2 D3
```
So we put the file handle into register D1, the address of our data buffer into register D2, the size of the buffer into D3, and call DOS function Read. When it returns, we’ll have the number of bytes actually read stored in D0. If this value is less than D3, we’re at the last block.

During the next section of code, we’ll refer to the address of the buffer several times, as well as the address of the output buffer. To save code, we’ll hold those addresses in registers A4 and A3 respectively. Let’s read some data.

```assembly
ReadBlock
    move.1 d6,d1 ; input file handle
    lea BufAdr(pc),a4 ; input buffer address
    lea OutBuf(pc),a3
    move.1 a4,d2 ; copy to D2 for read
    moveq BufSize,d3 ; input buffer size
    jar _LVORead(A6) ; so read it
    move.1 d0,d5 ; size of data

We have data. It’s in the buffer (whose address is held in A4). Time to look at it, count, and output. Register D4 will walk through the buffer area; we won’t let it pass the value in D5, which is the end of the buffer. After the comparison, the BCC (branch carry clear) command acts as a ‘branch greater than or equal to (unsigned)’. If you know the 6502, this might seem to be backwards; the carry flag is handled differently in the 68000 world.

; Here’s where we scan and output
moveq $0,d4 ; start of buffer pointer
; Character loop; are we at end of buffer?
NextChar
    cmp.l d5,d4
    bcc.s EndBuf

; We have data! Let’s crunch it.
move.b $01a4,d4),d0

Register d0, lowest byte, now holds the character from the buffer. We’ll bump the D4 pointer and then start to look at the character: is it a NewLine? is it non-printing?

```assembly
addq.l #1,d4
    comp.b #$0a,d0
    beq.s Nulliner

; Here’s where we scan and output
moveq $0,d4 ; start of buffer pointer
; Character loop; are we at end of buffer?
NextChar
    cmp.l d5,d4
    bcc.s EndBuf

; We have data! Let’s crunch it.
move.b $01a4,d4),d0

We compare the character to number 31, hex 1F. Note that BGT (Branch Greater Than) will branch if D0 is greater than 31. It’s a signed comparison, and we need to talk for a moment about that.

Signed numbers

When your cassette or video player counter shows 9999, you understand it to (usually) mean a true value of -1. The concept is this: very high numbers can be taken as being negative values if you make the rules work right.

If we ask the computer to treat a value as signed, as opposed to unsigned, we do exactly this. The rule works this way: if the high bit of the number in question is set, we will consider the entire number negative.

If we choose to use this rule, one-byte values don’t run from 0 to 255; instead, they go from -128 to +127. The computer handles the numbers in the same way, but comparisons are done in a different manner for signed numbers.

Consider our situation: we wish to reject any character with an ASCII value less than 32 decimal. Those are non-printing characters, of course. Consider: the normal ASCII character set stops at number 127. What are those others with higher values? Some are control characters, and others make up the ‘alternate’ character set. A program such as DUMP really doesn’t want these extra characters printed.

We could eliminate these extra characters (change them to a dot) by a second test; but we have a better choice. If we choose to consider these characters as if they were signed, all these 128-and-over characters become negative. A signed comparison will throw them out immediately.

In fact, it’s not the comparison that is signed; it’s the way we test following the comparison. And BGT does just what we want; special characters above the normal ASCII range not still be printed. We skip over the next (‘dot-making’) instruction.

It takes a while to get used to which way round the comparisons go. If we do not have a character of 32 or over, we change it to a dot (hex 2e, decimal 46).

Whether we have changed it or not, we print (‘Write’) the character and increase the line count by one. Remember that we’ve put the address of the tiny output buffer in register A3, which saves us some work here.

```assembly
NotDot
    move.b d0,(a3)
    move.1 a5,d1 ; file handle
    move.1 a3,d2 ; output buffer
    moveq $1,d3 ; length
    jar _LVOWrite(A6) ; count and send NewLine if needed
    addq.d #1,d7 ; bump column count
    comp.b #$64,d7 ; at limit?
    bls.s NextChar
```
If the number of characters has gone over 64 (a value I picked for no special reason), we will not take the above branch (BLS for Branch Less than). Instead, we'll drop through to NuLine and start a fresh line. We also come here if we see a NewLine character in our file.

Nutiner

move.b #80a,(a3)
move.l a5,d1 ;file handle
move.l a3,d2 ;buffer
moveq #1,d3 ;length
jar _LVWrite(A6)

In a moment, we’ll loop back to get the next character and continue our file dump. But first, a message from CTRL-C... maybe.

We detect CTRL-C by calling an Exec function, SetSignal. It’s not documented too well in the Kernel manual, but the following code does the trick.

First, keep in mind that we are about to call the Exec library and thus will need a pointer to ExecBase in register A6. We’d better save our DOSBase pointer by tucking it into register 07.

move.l a6,d7 ;stash DOSBase
moveq 60,d0
move.l 151000,d1
movea.l $4,a6
;jar _LVOSetSlgnal(A6);test CTRL-C
move.l d7,a6 ;restore DOSBase

We set up our CTRL-C mask in D1 and call SetSignal, then restore the DOSBase pointer as quickly as we can.

Now, let’s analyse what results have come back in register D0. We extract a single bit using the AND operation. If you don’t know AND, think of it as ‘remove all the bits except the one I’m interested in’.

andi.l #81000,d0
tst.1 d0
beq.s NotCtrlC

If D0 contained zero after the AND masking, we can breathe a sigh of relief and go back to our file reading. If not, we’ve detected CTRL-C and should stop the program.

Notice that this is neat ... the stop action takes place only at the end of a line. No broken lines to mess things up. At the start of a new line, we’ll print a ‘control-c’ type of up-arrow, C, and Newline to signal the stop. Then we’ll stop.

move.l a5,dl ;file handle
lea CtrlCMesg(pc),a5 ;message to
move.l a6,d2 ; buffer
moveq #3,d3 ;length
;jar _LVWrite(A6)

It seems odd, but putting a zero into register D5, our buffer size, will stop the program. We’ll connect this up in a moment.

moveq #0,d5 ;zap buffer causing exit

NotCtrlC
moveq #0,d7 ;Column count = 0
bra.s NextChar

We come to routine EndBuf when we have used up the buffer and want to fill it up again so that we can continue with the file. How do we know when to stop? There was a hint in our discussion of Read.

If we are not at the end of a file, we get a full buffer. Only at the end of the file is the buffer partly full. Now: we logged the number of characters we got into D5. If it’s equal to the buffer size, we may go back and do more. If not, we have finished the file. Print a NewLine to make things neat and quit.

Mental exercise: what happens if the last block exactly fills the buffer?

; The buffer is used up. If full, refill it.
EndBuf
cmp.w #BufSize,d5 ; was it a full buffer?
beq ReadBlock ; yes, get more
move.b #80a,(a3) ; no, send NewLine
move.l a5,d1 ; file handle
move.l a3,d2 ; buffer
moveq #1,d3 ; length
;jar _LVWrite(A6)
rts ; job complete

Our fixed data is just above: the DOS library name and the CTRL-C echo message. Coming up next: our two buffers. The output buffer is only one character long, and the input buffer ten characters. DC.B means ‘put a zero here’; DS.B means, ‘set aside ten characters, but I don’t care what’s there’.

OutBuf dc.b 'control-c'
BufSize EQU 10
BufAdr dc.b BufSize

The assembler directive EQU (equate) generates no code or memory. It just says, ‘any time you see BufSize, take it to mean value 10’.

That’s it. Assemble and link it, and you’ll find you have a handy little file snooper.

Impure thoughts
The above code is not pure. If the program were trying to do two (or more) things at once, the two buffers would probably get the data mixed.
This would not happen if we just opened two separate CLI windows and commanded `DUMP <filename>` in each. A separate copy of program `DUMP` would be loaded for each job, and the would be no conflict. The conflict will arise if we make `DUMP resident`.

Assuming you have Workbench 1.3 or later, try commanding `RESIDENT DUMP PURE` (PURE will override the fact that we have not set the ‘pure’ bit on the program file). Now, open two Shell windows ... only Shell windows use Resident commands. In each window, type `DUMP <filename>`. The file names may be the same or different; I would suggest text files in RAM, such as your .ASM source files. They are easy to read, and you can spot any problems quickly.

One copy of `DUMP` in memory tries to run both files, and, as you will see, fails miserably. There’s no danger to the system in this case, but you will see that the output is a mess.

It’s only the buffers that cause this effect. Is there an easy fix? Let’s get our buffers from the stack.

**LINK and UNLK**

If the program commands `LINK A4, 8-$12`, 16 bytes (hex 10) will be taken from the stack and reserved for your use. Register `A4` will point at the top of this memory area; we can reach the memory by using an offset from `A4`.

We must be absolutely sure to return the borrowed memory when we are finished with it, using `UNLK A4`. Because of the way the stack works, we must return the memory in the same subroutine that asked for it.

A detailed explanation of how the register and stack work together with commands `LINK` and `UNLK` would take up too much space here. But it’s a handy and dependable way of getting a modest amount of memory. See Figure 1 for an illustration of what happens.

**A Pure program**

This is similar to the earlier version of `DUMP`. Only the new code will be commented.

```asm
; Dump pure - Jim Butterfield March 16/99.
; Exec library calls

; xref LV0OpenLibrary ;$23
; xref LV0CloseLibrary ;$19
; xref LV0SetSignal ;$13

; DOS library calls

; xref LV0Open ;$18
; xref LV0Close ;$24
; xref LV0Output ;$3C
; xref LV0Write ;$30
; xref LV0Read ;$2A

bufsize EQU 10

;-- Initial setup:

move.1 a0,a4 ; save pointer to argument line.
move.1 d0,d4 ; save length of argument line.
lea dosname(po),a1 ; 'dos.library'.
```

![Figure 1: LINK, used with any A register causes a frame of memory to be allocated by the stack. UNLK returns the memory to the stack, and restores the former value in the A register. A7 is the Stack Pointer - sometimes called SP.](image-url)
EndBuf

We have data! Let's crunch it.

$0(a2,d4),d0 \quad \text{lt} 1,d4

NuLiner

NotDot

d0, (a3)

a5,d1 \quad ;file handle

move.1 a3,d2 \quad ;outbuffer

moveq #1,d3 \quad ;length

jst _LVOWrite(A6) \quad ;count and send NewLine if needed

addq.b #1,d7 \quad ;bump column count

cmp.b #$64,d7 \quad ;at limit?

bls.s NextChar

NuLiner

move.b #$0a,(a3)

move.1 a5,d1 \quad ;file handle

move.1 a3,d2 \quad ;outbuffer

moveq #1,d3 \quad ;length

jst _LVOWrite(A6)

move.1 a6,d7 \quad ;stash DOSbase

moveq #0,d0

move.1 #$4000,d1

move.1 $04,a6

jst _LVOSetSignal(A6); test CTRL-C

move.1 d7,a6 \quad ;restore DOSBase

and.1 #$1000,d0

tst.1 d0

beq.s NotCtrlC

move.1 a5,d1 \quad ;file handle

lea CtrlCMess(pc),a0 \quad ;message to

move.1 a0,d2 \quad ;buffer

moveq #3,d3 \quad ;length

jst _LVOWrite(A6)

moveq #0,d5 \quad ;zap buffer causing exit

NotCtrlC

moveq #0,d7 \quad ;colcount = 0

bra.s NextChar

; The buffer is bare. If twas full, refill it.

EndBuf

cmp.w #BufSize,d5 \quad ; was it a full buffer?

beq ReadBlock \quad ;yes, get more

move.b #$0a,(a3) \quad ;no, send NewLine
move.1 a5,d1 ; file handle
move.1 a3,d2 ; buffer
moveq #1,d3 ; length
jar _IVOWrite(A6)
rts ; job complete
dosname dq.b 'dos.library',0
CrlCMess dc.b ,AC 1 ,40a
end

Try this one as a resident program. You'll find it to be pure as the driven snow.

Commands encountered so far

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Addressing Modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD &lt;ea&gt;,Dn</td>
<td>Add binary</td>
<td>[B., W., L]</td>
</tr>
<tr>
<td>ADD Dn,&lt;ea&gt;</td>
<td>Add binary</td>
<td>[B., W., L]</td>
</tr>
<tr>
<td>ADDA &lt;ea&gt;,An</td>
<td>Add to A reg</td>
<td>[W., L]</td>
</tr>
<tr>
<td>ADDI #&lt;data&gt;,&lt;ea&gt;</td>
<td>Add immediate</td>
<td>[B., W., L]</td>
</tr>
<tr>
<td>ADDQ #&lt;data&gt;,&lt;ea&gt;</td>
<td>Add quick</td>
<td>[B., W., L]</td>
</tr>
<tr>
<td>AND &lt;ea&gt;,Dn</td>
<td>AND logical</td>
<td>[B., W., L]</td>
</tr>
<tr>
<td>AND Dn,&lt;ea&gt;</td>
<td>AND logical</td>
<td>[B., W., L]</td>
</tr>
<tr>
<td>ANDI #&lt;data&gt;,&lt;ea&gt;</td>
<td>AND quick</td>
<td>[B., W., L]</td>
</tr>
<tr>
<td>BEQ</td>
<td>Branch Equal</td>
<td>[opt. S]</td>
</tr>
<tr>
<td>BNE</td>
<td>Branch Not Equal</td>
<td>[opt. S]</td>
</tr>
<tr>
<td>BGE</td>
<td>Branch (signed) Greater/Equal</td>
<td>[opt. S]</td>
</tr>
<tr>
<td>BLT</td>
<td>Branch (signed) Less Than</td>
<td>[opt. S]</td>
</tr>
<tr>
<td>BGT</td>
<td>Branch (signed) Greater Than</td>
<td>[opt. S]</td>
</tr>
<tr>
<td>BLR</td>
<td>Branch (signed) Less/Equal</td>
<td>[opt. S]</td>
</tr>
<tr>
<td>BRA</td>
<td>Branch Always</td>
<td>[opt. S]</td>
</tr>
<tr>
<td>BSR</td>
<td>Branch Subroutine</td>
<td>[opt. S]</td>
</tr>
<tr>
<td>CLR &lt;ea&gt;</td>
<td>Clear to zero</td>
<td>[B., W., L]</td>
</tr>
<tr>
<td>CMP &lt;ea&gt;,Dn</td>
<td>Compare</td>
<td>[B., W., L]</td>
</tr>
<tr>
<td>CMPI #&lt;data&gt;,&lt;ea&gt;</td>
<td>Compare Immediate</td>
<td>[B., W., L]</td>
</tr>
<tr>
<td>JSR &lt;ea&gt;</td>
<td>Jump Subroutine</td>
<td>[opt. S]</td>
</tr>
<tr>
<td>LEA &lt;ea&gt;,An</td>
<td>Load Effective Address</td>
<td>[L only]</td>
</tr>
<tr>
<td>LINK An,#&lt;displacement&gt;</td>
<td>Link and Allocate</td>
<td></td>
</tr>
<tr>
<td>MOVE &lt;ea&gt;,&lt;ea&gt;</td>
<td>Move data</td>
<td>[B., W., L]</td>
</tr>
<tr>
<td>MOVEA &lt;ea&gt;,An</td>
<td>Move Address</td>
<td>[W., L]</td>
</tr>
<tr>
<td>MOVEQ #&lt;data&gt;,Dn</td>
<td>Move Quick</td>
<td>[L only]</td>
</tr>
<tr>
<td>RTS</td>
<td>Return from Subroutine</td>
<td></td>
</tr>
<tr>
<td>SUB &lt;ea&gt;,Dn</td>
<td>Subtract binary</td>
<td>[B., W., L]</td>
</tr>
<tr>
<td>SUB Dn,&lt;ea&gt;</td>
<td>Subtract binary</td>
<td>[B., W., L]</td>
</tr>
<tr>
<td>SUBA &lt;ea&gt;,An</td>
<td>Sub from A reg</td>
<td>[W., L]</td>
</tr>
<tr>
<td>SUBL #&lt;data&gt;,&lt;ea&gt;</td>
<td>Sub immediate</td>
<td>[E., W., L]</td>
</tr>
<tr>
<td>SUBQ #&lt;data&gt;,&lt;ea&gt;</td>
<td>Sub quick</td>
<td>[E., W., L]</td>
</tr>
<tr>
<td>TST &lt;ea&gt;</td>
<td>Test, compare to zero</td>
<td>[B., W., L]</td>
</tr>
<tr>
<td>UNLK An</td>
<td>Unlink</td>
<td></td>
</tr>
</tbody>
</table>

Addressing modes encountered so far

- **Dn** Data Register Direct.
  - Contents of the data register.

- **An** Address Register Direct.
  - Contents of the address register.

- **(An)** Address Register Indirect.
  - Contents of the memory addressed by the address register.

- **(An) +** Address Register Indirect with Postincrement.
  - As above, but add to the address register after use.

- **(An) -** Address Register Indirect with Predecrement.
  - As above, but subtract from the address register before use.

- **x (An)** Address Register Indirect with Displacement.
  - Contents of the memory addressed by: the address register, with this address adjusted by a displacement ranging from -32768 to +32767.

- **x (An), Dm** Address Register Indirect with Index.
  - Contents of the memory addressed by: the address register contents, plus the Data register (Word or Longword), plus a displacement ranging from -128 to +127.

- **x** Absolute Address.
  - The address as supplied in the program.

- **x (PC)** Program Counter with Displacement.
  - The 'current location' of the program, plus a displacement ranging from -32676 to +32676.

- **x (PC), Rn** Program Counter with Index.
  - The 'current location' of the program, plus the Data or Address register (Word or LongWord), plus a displacement ranging from -128 to +127.

- **#x** Immediate.
  - An actual value to be used.
Bon voyage to Glen McDiarmid, one of Brisbane's favourite sons. Glen has gone on extended leave/R&R/rec leave or what you will to the USA. Given Glen's talents, we hope his leave is not too extended, but we wish him well in trying to establish himself whether he decides to stay or not. He did mention that he was going to check out the software, but whose he didn't say.

I was talking to another Brisbaneite the other week, Greg Perry. The conversation was about ANSI codes—the printer codes that supposedly allow you to send direct commands for printer control. I was trying to use them on screen, and eventually found a way of doing it. Then I tried to use them on my printer, only to find that most had no effect whatsoever. This had me puzzled until I learned that a lot don't work with WBI.3. As well, there are quite a few errors and omissions in the Commodore listings provided in the manuals.

Asking around, I learned that if I used WBI.2 my problems would be over. Nearly. I2 certainly passes ANSI codes more consistently than 1.3, but the manual errors will still trip you up. My conclusion is—find another way. The double jeopardy of using the codes is not worth the paper and candlewax.

One other point, before getting serious. If you have an A500 with A501 expander, and are losing lumps of memory from your machine from time to time, there is a possible fix. The power supply pins on Rev5 main boards have a very small heat dissipation area. The get hot, things come loose, and the A501 loses volts. Visually, there will be a blackening around the pin area (this is nearest the front of the bunch of pins that connect to the A501). The solution is—resolder around the pin landings, providing a good coating as you do so.

Word Perfect has not shut down—merely gone into very low gear. The planned major rewrite remains cancelled, but some of the bugs in the present version will be swatted. For a very high priced package, that doesn't seem very appealing, does it?

Received a pleasant note from Nev Beatty at NWAUG in Melbourne. If you live in the north-west of Melbourne, you might care to trot along to one of their meetings. Contact Nev at PO Box 204, Essendon 3040.

Also in the mail was a letter and disk from Foundation HARD-USER International (PO Box 1057, NL-5602 BB Eindhoven, The Netherlands). Apart from their magazine on a disk, they are busy selling all sorts of Amiga add-ons. To quote their pricelist—A501 for £126stg, Delux Paint III for £67stg, Delux View 3.5 for £107stg. They sound reasonable prices to me. They didn't give a phone number (who can afford to ring, anyway?) but they did give their bank account number. Odd European habit or an invitation to hackers?

Necessity is the mother of invention, and mailing out this magazine has taught me a trick or two about our postcode system, Australia Post being a tad backward in this area. If anyone wants some astonishing stuff about a boring but necessary chore, just talk to me.

We all eat our hearts out for better printer output from our Amigas, but most of us can’t/won’t contemplate the megabucks that it takes. 24 pin dot matrix printers do well (see the Queensland User Group magazine if you want proof of that statement) but I was looking at some Canon printers of the inkjet tribe, and was quite impressed by their performance. And their price, which was less than $2000. Is anyone out there using one? We would like to hear, because the mail we get suggests heavily that printer output from the Amy is probably the most troublesome item requiring resolution. Write, ring, but tell us, please.

Until the joystick port on my A500 fell apart, I had been spending a fair amount of time with Superbase Professional. I chose to do this because I was doing more and more ugly stuff with dBase IV on my office PC, and needed to be convinced that truth, light and ease of use still existed. The most vaunted database program in the world seems twice as difficult and multiple times more cumbersome to program than SBPro. That's probably showing my lack of erudition, but there are lots of me's out there, all of whom struggle from time to time. Long live Precision Software!

We need more authors (as every Editor screams into the midnight air!). Its hard to believe that the Amigan population out there has so little to say. It would be nice to run at least one Australian article per issue—if for no reason other than to tell the rest of the civilized world that we exist. Do computers breed illiteracy? Are we so busy playing with the darn things, or are we so selfish that we won’t share our experiences with others? Or do we have a bad dose of cultural cringe. Phew, that should get an angry letter or two.

But seriously folks (oh dear!), we will acknowledge your talents if you are prepared to do a bit of work. Theses are not required—just some short notes about what you like, think, feel or create.

And if you know of anyone thinking of writing and circulating a virus “just for the hell of it”, kindly strangle them. The one thing most likely to destroy Amiga credibility is a virus—any virus—even a hint of one. It’s just plain stupidity.

Next issue will be the last for 1989—then its into the 90’s. It promises to be an interesting time, especially in the Commodore camp. As one pundit put it—“it may not be the case of whither Commodore...”

Paul Blair
The UK Developers’ Conference

A personal view of the Nottingham conference

by Mike Todd

It’s a sobering experience to get amongst a large number of real Amiga programmers ... although ‘sobering’ is the wrong word, with free drink flowing until the early hours.

It started with the Amiga Developers Advisory Board (ADAB for short), set up some time ago in an attempt to improve liaison between Commodore UK and the many developers who felt that Commodore was just not listening to them. ADAB decided that, instead of just having meetings from time to time to moan at Commodore, they might do something constructive. So, led by chairman, Gordon Shields (of ASAP) and secretary, Dave Parkinson (of Ariadne software) the group decided to have a conference run by developers, for developers.

Although a cheap and cheerful conference, CATS(UK) and CATS(USA), the Commodore-Amiga Technical Support units, both provided equipment and moral support.

With this support came two visiting ‘celebrities’. The first was Carolyn Scheppner, whose name appears on many items (including IFF documentation) that come out of Commodore-Amiga. The other was Gail Wellington, General Manager, Worldwide Software and Product Support. Many will remember Gail from her days with Commodore UK and since those days, she has always treated the UK as her second home.

Between them, Gail and Carolyn gave the conference insights into the future of the Amiga, including news of new products and the long awaited 1.4 - for the news on 1.4, see page 18.

All I can do in the limited time and space available is give a brief outline of was was discussed. I can’t even guarantee attributing the information to the correct source. All I want to do is give a taste of what came out of a most interesting weekend.

Gail Wellington on technical support

Gail provides support for all Commodore products. Working for Commodore International in West Chester, she travels the world encouraging developers to develop - clearly believing that, without third party software and hardware, any computer product is but an empty shell.

As the Amiga has become more and more firmly established, it has been vital to provide improved support for those developing third party products. To provide this support, CATS (Commodore Amiga Technical Support) was set up, and is now expanding considerably.

In the US, there are four main support departments: The first, the Technical Group (headed by Carolyn Scheppner), has responsibility for all telephone and electronic (BIX and usenet) support, for the development of standards (from the user interface, to the IFF standard), developers’ material, documentation and so on.

The Non-technical Support Group handles marketing support and advice, distribution and liaison.

There is an Applications Development Group which takes on board any third party products whose development is considered vital to the progress of Commodore products. To help such projects, the group can share the risk by providing funds for product research, provided that these meet Commodore’s own targets. Such help has already been provided for an Amiga LOGO and an interactive authoring system.

The fourth group has been called the Evangelical Group, for want of a better title. It looks after developments in specialist areas, such as education, productivity, recreation and leisure.

In the US, two levels of support have been established. Certified developer status is open to anyone developing on or for the Amiga, whilst priority support will be given to those with commercial developer status.

If all this sounds a little remote, with the emphasis on the US operation, then fear not. Dr. Rahman Haleem, the UK’s Technical Support Manager, expressed his commitment to providing improved developer support, and he will continue to expand the UK support operation; indeed, he is currently looking for people of the right calibre to assist.

This support is being extended worldwide, eventually including an electronic link from a UK developers bulletin board, back to the Amiga development and support teams.
Rahman gave a feel for the scale of the operation when he reported that there are over 275 registered developers in the UK alone. With 30,000 telephone calls arriving at Commodore (of which 24,000 were technical queries), this might explain why, as most developers report, UK support has been pretty appalling; but with Gail’s and Rahman’s positive approach, I am sure we will see the situation improve.

Rahman admitted that he was jealous of the level of expertise outside of Commodore - expertise that reflected itself in the setting up of ADAB. In fact, the success of ADAB has prompted the US operation to consider something similar.

Continuing on the theme of development, Rahman pointed out that there are test facilities at Maidenhead. Anyone can come along and try out their product on almost every conceivable system configuration. This facility is available free to any developer and its use is to be encouraged.

Finally, on the subject of developer support, there is some pressure to charge for parts of the support program. At present this is free, but sadly this is going to have to change in the future.

The developers speak

Amongst the major developers who spoke to those gathered in a hot and stuffy lecture theatre was Dave Parkinson who talked about programming the Amiga in C and assembler. His talk reflected the fact that, while many of the gathered throng were very experienced Amiga programmers, there were many others there who were just getting to grips with the machine.

Simon Tmmer, whose Commodore programming career dates way back to the PET and Superscript, talked about working with Intuition. His talk developed into a discussion on the problems of the portability of software.

When large software houses develop software, it is now essential to be able to transfer the package across to other manufacturer’s machines with a minimum of fuss.

Superbase, Simon’s major software project, is a case in point. It is available for several other machines, including the IBM PC and to allow the company to maintain the package across the machine range, portability is essential. However, some of the Amiga’s unique features, such as Intuition screens, create problems - fortunately, Simon has been able to allow the Amiga version to use many of the Amiga’s features.

Another well known name from the early Commodore days is Harry Broomhall. Harry is noted for his early work inside the Commodore disk drives, and now flits around the country in a flash car as a consultant.

Harry stressed the point that AmigaDOS is fast, despite what many of us feel; it is capable of very fast data transfers, although there are overheads within the system that slow things down.

Apart from the speed, AmigaDOS seems capable of doing rather more than some other machines - Simon pointed out that, in an experiment, AmigaDOS was able to have open 1000 files and update every single one of them. UNIX is limited to about 64, while IBM PCs can only manage 20!

During Harry’s talk on AmigaDOS, the question of locks came up. It is essential that any software that asks for a lock (on a file or whatever) hands the lock back when it is finished - that is, it must unlock. Some system routines create a new lock for you, but others just give you access to a lock already held by the system and unlocking these will cause the system to fall apart - and no-one seemed sure which routines did which.

All this led to an important point about reliable software - make sure you deallocate all the resources that you used. Deallocate all your memory, close all libraries and devices that you opened, and unlock all your own locks.

Although there are several utilities to help you check if there are any locks left, or resources left allocated, it was suggested that there is a relatively easy method of checking the orderly termination of your software.

Run the software under WorkBench and, once the program terminates, take the disk out; the disk icon should disappear. If not, the chances are that you’ve left a lock behind. Also note the memory available before and after the program has been run - they should be the same, taking into account, of course, any soft-loaded libraries, window buffer space and so on.

Audio, of course, is a major Amiga feature and there was a talk by Kevin Stevens of DigiGraphics on the use of audio. Apart from explaining the how, why and wherefore of the techniques, two interesting comments were made.

The first relates to the problem of starting the playback of a second sample after stopping another mid-flight. This can’t normally be done and one would normally have to be patient and wait for the first to complete. However, Jez San (of whom more in a moment) suggested that setting the period to 1 will allow the new sample to be set up and retriggered.

The reasoning appears to be that the contents of the period register is used as a count-down register, and only when it reaches 0 is a new sample retrieved from the waveform data set. By setting the period to 1, the register reaches 0 very quickly and allows the new sample to be accessed.

The second discussion related to the quality of Amiga audio, the concepts of the Nyquist frequency (which specifies the minimum sampling rate required for a given upper frequency of the sampled waveform) and the filtering of Amiga audio.

The fudging of 14-bit audio was also discussed. By using the 6-bit volume control as the most significant bits of the sample, and the 8-bit DAC to provide the least significant bits, it was claimed that CD quality could be available. However, this
overlooked the non-linearity of the Amiga’s volume control and the lack of precision in the system. Also, such audio would have to be driven by software rather than rely on DMA, and the results would be non-simultaneous presentation of the two sections of the audio sample.

New products

Gail was able to describe the continuing development of Commodore’s Amiga products, and the future looks exciting.

There has already been a great deal of talk about the A2500UX machine - a UNIX based machine which will autoboot UNIX System V, version 3.1. Inside are two of the new cards Gail described; an A2630 and the A2091.

The A2630 is a 68030 card (running at 25 MHz), including 68882 co-processor, memory management, 2 or 4 Mbyte of 32-bit RAM and the ability to accept a daughter board.

The A2091 is a hard disk controller board with on-board SCSI interface. It is capable of auto-booting, and has a new DMA chip to solve some of the DMA contention problems that can arise. This board is expected to be available in a few months and will be shipped with the UNIX box complete with 100 Mbyte, 19 millisecond hard disk and a 150 Mbyte tape-streamer.

Commodore appears keen to improve the video and graphics capabilities of the Amiga, and to this end there is the A2024, a high resolution (1008x1024) monochrome monitor for use on any Amiga with more than 1 Mbyte of RAM. 4 shades of grey and full horizontal overscan are supported. While the PAL model has already been released, the NTSC model has been held up awaiting FCC approval.

Colour hasn’t been forgotten, and a new hi-res colour graphics card, the A2410, is being developed. Based around a TI graphics processor, it should have a 1024x768 non-interlace display, with a palette of 256 colours out of 16 million - and will support TIGA. It will work with UNIX and Amigados, although its output will be dedicated and a second monitor will be needed for the main system screens (such as Workbench).

To compliment the output devices, a Professional Video Adapter (the A2350) is being considered. This will be a real-time frame grabber and genlock unit, and a new paint package has already been developed that works on the image within the PVA’s frame store.

A500 owners haven’t been forgotten, with a workhorse hard disk drive, the A590+. This has a 20 Mbyte drive, capable of accepting up to 2 Mbyte of fast RAM, a SCSI connector on the back and the new DMA chip described earlier.

Talking of new chips, the new enhanced chip set (ECS) is now starting to emerge - giving new graphics modes - 640x480 (non-interlaced - thus no flicker, but requiring a multi-sync monitor) and 1280x512 (interlaced) using 4 colours from 64.

The new Fat Agnus chip is already available, and A2000s with revision 6 boards are now being shipped with it. This chip gives access to a full 1 Mbyte of chip RAM, and the potential for 2Mbyte in the future. However, the chip defaults to NTSC mode, and without 1.4 there’s not a lot we can do about this other than bend back one of the pins to force it into PAL mode. This isn’t made clear in the installation notes!

The new Denise chip is still in its test phase. However, since it requires version 1.4 of the operating system, we’d have to wait to use it anyway.

Jez San - hitting the hardware

Jez San, well known games writer and hitter of the hardware, talked about hitting the hardware legally. A contradiction in terms? Well, no - not if you do it properly. For instance, if you want to play with the parallel port, then claim the resource first so that no-one else can use it. Then you can ‘hit the metal’.

Some games need all the speed and memory that they can get and so need to have full control of the system. Jez insists that, provided that the machine is taken over in an orderly and sensible manner then there is no harm, while others would argue against, on the basis that there is no way back to the system under these circumstances other than re-booting.

It was interesting at this point to hear Gail accept some of the points made and express an enthusiasm for finding a method of doing all this legitimately.

Carolyn reminded those that do access the hardware registers of two fundamental points. Write-only registers are just that. Read them and you will write random data.

Do not ignore unused bits, since they may not be unused on the ECS. If reading, mask out unwanted bits; you can no longer assume they will be zeroes; if writing, always put zeroes in unused bits since spurious ones may have undesired side effects.

For those doing large blits for games, Jez suggests that these be done just as vertical blanking starts. At this point, no screen data is being gathered and the blitter will work twice as fast.

Finally

Well, that’s the first ADAB developers’ conference over - a conference with a difference. It was non-profit-making, and the small excess of income over costs was put behind the bar on the Saturday night. Many took advantage of this and the bar extension, and stayed drinking until 1 a.m.

It was an informal conference, with several good speakers (although not all used to speaking in public). The atmosphere was solid Amiga, and everyone seemed to learn something new.

With over a hundred delegates passing through, it was a significant gesture of confidence in the Amiga here in the UK.
The Amiga Interrupts

A cool guide to the interrupts and how to use them

by Ian Potts
Copyright © 1989 CoolSoft

Ian Potts is a freelance programmer who runs a software development house. He has been programming for 7 years and is responsible for several commercial games for the Commodore 64 and Amiga. Ian is currently working on a new entertainment product for the Amiga, and a word-processing package. He is one of the few Amiga games programmers who uses the Operating System legally, rather than just 'hitting the hardware'.

Hardware support for interrupts

The Amiga's CPU (the 68000) has seven levels of interrupts, known as exceptions. These range from level 1 up to level 7, the Non Maskable Interrupt. As with other 68000 exceptions, vectors for the interrupts are stored in the first 1K of the Amiga's memory from $564 to $57C. When an interrupt occurs, the computer pushes the status register and program counter onto the current user stack and jumps to the appropriate interrupt code via one of these vectors.

The seven interrupt levels are prioritised so that a higher level interrupt can interrupt a lower level interrupt but a lower level one can not interrupt a higher level interrupt. Thus more important interrupt sources are set to the higher level interrupts.

In the Amiga the Paula chip takes things a bit further as it actually provides fourteen levels of interrupts rather than seven. These fourteen interrupts are mapped onto the CPU's interrupts levels so that several interrupt sources can share the same CPU interrupt level as shown in figure 1.

The Non Maskable Interrupt (NMI) can be generated by external hardware and generates a level 7 interrupt with its vector at $7C.

When Vertical Blank occurs (when the video beam reaches the bottom of the display, and starts its journey back to the top) a level 3 interrupt is generated, the processor stops what it is doing, saves the status register and program counter to the user stack, and jumps to the routine whose address is stored in $6C. It then runs this routine which is terminated by an RTE (ReTurn from Exception) instruction, which restores the program counter and status register and lets the processor continue.

However the level 3 interrupt routine can be called by a Copper, Vertical Blank or Blitter Finished interrupt, so the routine must be able to test which interrupt has occurred and act appropriately.

This is done by reading Paula's interrupt registers. These are:

<table>
<thead>
<tr>
<th>Interrupt Register</th>
<th>Description</th>
<th>Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>INREQ</td>
<td>Interrupt Request Bits (Clear or Set)</td>
<td>$DFF09C</td>
</tr>
<tr>
<td>INREQR</td>
<td>Interrupt Request Bits (Read)</td>
<td>$DFF01B</td>
</tr>
<tr>
<td>INTENA</td>
<td>Interrupt Enable Bits (Clear or Set)</td>
<td>$DFF09A</td>
</tr>
<tr>
<td>INTENAR</td>
<td>Interrupt Enable Bits (Read)</td>
<td>$DFF01C</td>
</tr>
</tbody>
</table>

Setting or clearing bits in INTENA enables or disables the appropriate interrupt source. The bits correspond to those shown in figure 1; that is bit 0 is TBE, bit 3 is VERTB and so on.

Bit 14 (INTEN) is the master interrupt enable bit. If this is cleared, the ROM's interrupt routines will all exit immediately without running any more of the interrupt code.

<table>
<thead>
<tr>
<th>Paula Interrupt Source</th>
<th>CPU Interrupt Level</th>
<th>CPU Interrupt VECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBE</td>
<td>Transmit Buffer Empty</td>
<td>1</td>
</tr>
<tr>
<td>DSKBLK</td>
<td>Disk Block Finished</td>
<td>1</td>
</tr>
<tr>
<td>SOFT</td>
<td>Software Interrupt</td>
<td>1</td>
</tr>
<tr>
<td>PORTS</td>
<td>I/O Ports &amp; Timers</td>
<td>2</td>
</tr>
<tr>
<td>COPER</td>
<td>Copper</td>
<td>3 or 6</td>
</tr>
<tr>
<td>VERTB</td>
<td>Start of Vertical Blank</td>
<td>3</td>
</tr>
<tr>
<td>BLIT</td>
<td>Blitter Finished</td>
<td>3</td>
</tr>
<tr>
<td>AUD0</td>
<td>Audio Channel 0 Finished</td>
<td>4</td>
</tr>
<tr>
<td>AUD1</td>
<td>Audio Channel 1 Finished</td>
<td>4</td>
</tr>
<tr>
<td>AUD2</td>
<td>Audio Channel 2 Finished</td>
<td>4</td>
</tr>
<tr>
<td>AUD3</td>
<td>Audio Channel 3 Finished</td>
<td>4</td>
</tr>
<tr>
<td>RBF</td>
<td>Receive Buffer Full</td>
<td>5</td>
</tr>
<tr>
<td>DSKSYN</td>
<td>Disk Sync Register Match</td>
<td>5</td>
</tr>
<tr>
<td>EXTER</td>
<td>External Interrupt</td>
<td>6</td>
</tr>
<tr>
<td>INTEN</td>
<td>Master Interrupt Enable</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 1: Amiga interrupts
To set or clear bits in INTENA, write a 1 into the bit or bits required with bit 15 (SET/CLR control) set to 1 to set the bits, or 0 to clear them.

The INTENA register lets you read the current status of the interrupts. The INTREQ and INTRBQR registers allow you to either cause an interrupt by setting bits in INTREQ, or to test which interrupts have occurred by reading INTRBQR.

Figure 2 shows the routine in the Amiga's ROM that is called via the Level 3 vector at $6C.

The routine first saves several registers to the stack. It then tests whether the Master Interrupt Enabled bit (INTEN) is set in Paula's intena (SDPFO1C) register. If it is clear, it exits the routine as all interrupts have been disabled. Otherwise it ANDs the value from INTENA with the contents of INTREQ (SDPFO1E) which contains the bits of the interrupts that have occurred. The routine then goes through testing which bit is set; if bit 6 (Blank Interrupt) is set it calls the Blitter Finished Interrupt routine, if bit 5 is set it calls the Vertical Blank Routine and if bit 4 is set it calls the Copper Routine. If none of these interrupts have occurred, the routine exits.

Each of the other hardware interrupt level vectors call similar routines which test the bits in Paula to decide which interrupt has occurred and act correspondingly.

Interrupt vector structures

Figure 2 shows that, when the level 3 interrupt routine in the Amiga's ROM has decided which Paula interrupt occurred it then runs three main instructions. In the case of the Vertical Blank interrupt these are,

NOVL.L $8A(A6),AI/A5 ;IVBLIS: IV DATA into AI, IV CODE into A5
PEA -$24(A6) ;Exit Interrupt Vector structures.

At this point, A6 contains the address of the ExecBase structure (described in the exec/execbase.i include file). The first instruction references the IVBLIS Interrupt Vector structure in ExecBase. This structure is defined in the exec/interrupts.i include file as follows:

```
STRUCTURE IV,8
  AIPTR IV_DATA ;address of data to be passed to interrupt routine
  AIPTR IV_CODE ;address of interrupt routine to be called
  AIPTR IV_DATA ;address of the interrupt server/node structure that was used to install this Interrupt Vector when SetIntVectcr was called
  LABEL IV_SIZE ;size of the interrupt vector structure
```

This structure is a private Exec structures used to jump to interrupt routines. [private structures are just that, and are for the system's own use. Anything that is private can be expected to change in future releases, such as 1.4 ED]
In the first instruction above, the IV_DATA field of the structure is moved into A1 and the IV_CODE field is moved into A5. The IV_CODE field contains the address of the routine to be jumped to when an interrupt occurs and the IV_DATA field contains the address of data which can be accessed by that routine. In the case of Vertical Blank, the IV_CODE points to the Interrupt Server Chain calling routine in Exec (see figure 3) and the IV_DATA points to a list header node which heads a list of server routines that should be called during Vertical Blank.

The second instruction pushes the address of the Exec routine Exit/int vector onto the stack. This routine will be called when the interrupt vector code has finished. It is a private Exec routine, called after every interrupt, which essentially controls the multitasking ability of the Amiga by switching between tasks after a given amount of time.

The third instruction then jumps to the routine pointed to by A5, which is the IV_CODE address from the interrupt vector structure. In the case of Vertical Blank (and several other interrupts, like the Copper) this will jump to the routine in figure 3.

As we have seen, there are two types of interrupt vectors on the Amiga: the hardware vectors ($64-$7C) and Exec's vectors contained in ExecBase. In general you should never alter the hardware interrupt vectors as this will stop multitasking. For applications that take over the whole machine (like games) this can be acceptable, but is usually unnecessary.

The interrupt vector structures in ExecBase can be altered to jump to your own interrupt routines by calling an Exec routine, SetIntVector.

This is called as follows:

oldinterrupt = SetlntVector(intNumber, interrupt)

The interrupt routine that you install with SetlntVector should clear the interrupt request bit in INTREQ (S5F09C) which caused the interrupt before it exits.

For example, if we have used SetlntVector to alter the Vertical Blank interrupt vector, our routine should exit like this:

move.W 0 INT_VEC,customintreq ; clear INT bit in intreq
rsr .w ; always exit with an RTS instruction

For an example of how to use SetlntVector routine see SetIntVecEx.asm in listing 2.

Note that you should never alter Exec's interrupt vector structures directly, always use SetlntVector.

Interrupt servers, and server chains

In a multitasking system like the Amiga, you need a better system for handling interrupts than just vectors. This is because several tasks may want to set a particular vector (such as the Vertical Blank vector) to point to their own interrupt routines. A vector can only point to one routine at a time so that tasks would steal the vector from other tasks, causing havoc.

This is why the Amiga supports Interrupt Server Chains. These allow several tasks to share interrupts by adding interrupt servers to server chains.

When an interrupt occurs, all the servers on a given chain will be run in order, so all the tasks will get to use the same interrupt.

The main interrupts which use server chains are the Vertical Blank and Copper interrupts. In the code shown in figure 2, these interrupts will jump to the routine shown in figure 3 via their interrupt vectors. This stops through the interrupt server chains for these interrupts, calling each server in order until it reaches the end of the chain, or until a server returns a non-zero value thus terminating the chain.

The routine is called with A1 pointing to the server chain header structure. This structure is a private exec structure [and therefore, as mentioned before, cannot be relied on ED]. The main information contained in it is the first long word of the structure which contains the address of the first interrupt server structure in the chain. The routine then tests whether that server has N SUCCESSOR; if not it exits, otherwise it puts the IS_DATA field into A1 and the IS_CODE field into A5 and calls the server routine.

On return, if DO returned non-zero the routine exits, otherwise it gets the address of the next server in the chain from LN_SUCCESSOR and loops.
To add interrupt servers to a chain we use the Exec routine, `AddIntServer` as described:

```
AddIntServer(IntNumber, interrupt)
```

`IntNumber` (D0) contains the number of the Paula interrupt bit for the server chain you want to add a server to.

`interrupt` (A1) contains the address of your interrupt server structure.

Server structures are inserted into the chain depending on the value of `INyk7` in their structure. Higher priority servers will be added near the top of the chain and will be run first.

Listing 1, `AddIntSrvEx.asm`, shows an example of adding an interrupt server into the Vertical Blanking chain to move a sprite around the screen.

When adding an interrupt server to the Vertical Blank chain, never make the priority (`INyk7`) equal to or larger than 10 because the Graphics Library has an interrupt server of priority 10 which reloads the copper list each frame; if it is delayed, the copper list won’t get loaded until the next frame which can mess up double buffering of graphics.

Interrupt server routines are called with the following conventions:

- `d0` - scratch
- `d1` - scratch
- `a0` - scratch
- `a1` - server data segment pointer - `IS_DATA` (scratch)
- `a5` - scratch
- `a6` - scratch

All other registers must be preserved. Some documentation indicates that `A6` contains the address of ExecBase when calling servers. This is only correct for the first server in the chain.

If a server alters `A6`, the next server will receive the altered value in `A6` as the routine in figure 3 does not reload `A6` with ExecBase between calling each server. Thus you should always load `A6` with ExecBase yourself if you want to access Exec from within your server.

When your interrupt server has finished, it should return 0 in `D0` to allow other servers on the chain to run. Returning non-zero in `D0` will terminate the chain.

System software interrupts

If you examine the data structures that make up the vertical blank interrupt server chain, you will find that there are three server routines on it as standard. The first is the Graphics Library’s server. This essentially handles loading the copper with the current copperlist either for a non-interlaced screen (same list each frame) or an interlaced one (alternates between `gb_LOFlist` and `gb_SHFlist`), and signalling tasks that have called WaitTOF.

The next server called is the `gameport.device` server, which reads the gameports after a certain number of frames.

The final server in this chain is the `timer.device` server which controls the VBlank timer device timing.

The Copper server chain initially has no servers in it and only springs to life when a task adds a server to it and sets up a copperlist which causes a copper interrupt by setting the `COPER` bit in `INTREQ ($DFF90C)`.

The Blitter Finished interrupt doesn’t run a server chain. Instead, it calls a routine which handles queued Blitter requests set up by the Graphics Library routine `QBSBlit`.

The audio channels interrupt vectors are not set up fully until either the audio device or some program does `SetIntVector` on them. See `SetIntVecEx.asm` in listing 2 for an example.

The Disk Block Finished and Disk Sync Interrupts jump to routines in the `trackdisk.device` which controls disk reading and writing.

The Software Interrupt is a special interrupt which can be forced to occur by a program either setting the `SOFT` bit in `INTREQ ($DFF90C)` or calling the Exec routine `Cause`, which will cause a software interrupt to occur.

Putting it all into practice

Right, now that we have covered most of the theory about Amiga interrupts we shall take a look at actually using them in our own programs.

Listing 1, `AddIntSrvEx.asm`, demonstrates how to add an interrupt server to the Vertical Blank server chain.

The program is written in Assembler as this is the best language to use for interrupts because of its speed. The program moves a simple sprite up and down from within the interrupt.

Listing 2, `SetIntVecEx.asm`, demonstrates how to alter Exec’s interrupt vectors.

The audio block finished interrupt vectors are not set up properly on the Amiga until the audio device is used. Thus listing 2 sets up a vector to point to my `its` routine which switches off the sound DMA of an audio channel.

Both listings were written with HiSoft’s DevPac V2.08 and should provide a useful basis for writing your own interrupt routines. Have fun!
Listing 1: AddIntSrvEx.asm

; Arrange Interrupts. By Ian Potts. (c)1989 CoolSoft.

; Listing 1. Example of AddIntServer

; Assembled with DevPac V2.08 By HiSoft Ltd.

opt d,c,e,w     ;en debug, optimise es
includ "rad/include/*
include "exec/exec_lib.lib" ;exec
include "graphics/graphics_lib.lib" ;graphics
include "graphics/view.lib" ;fer view stuff
include "graphics/sprite.lib" ;fer sprite stuff
include "hardware/intbita.lib" ;int bit definitions
include "exe/interrupts.lib" ;server structure
include "misc/easystart.lib"

moveq $31,d0     ;open graphics library
lea graflib(pcl),a0
CALLEXEC OpenLibrary
move.1 d0, GfxBase

moveq $31,a0     ;open intuition library
lea Intuilib(pcl),a0
CALLEXEC OpenLibrary
move.1 d0, IntuitionBase

CALLINT ViewAddress ;get address of View
move.1 d0,a0
move.1 v ViewPort(a0),viewport ;and first viewport

bar sprite       ;get a hardware sprite
tac.1 d0         ;successful?
bne.a exit       ;no, sigh
move.b $10,alldone ;bounce sprite 10 times
move.1 myinterrupt,myinterrupt\$CODE
clr.b myint+15 DATA set up our
clr.b myint+c_PKI ;server structure
lea myint(pcl),a0
move.1 INT있는,DIR add it to Vertical
CALLEXEC AddIntServer ;start chain

senare: comp.b $0,alldone ;wait until bounced
bne.a sprite     ;10 times
lea myint(pcl),a0 ;remove sur server
move.1 INT있는,DIR ;find the chain
CALLEXEC RemIntServer

clr.b d0          ;free up sur sprite
move.w mysprite64,wxm,0
CALLEXEC FreeSprite

exit: move.1 IntuitionBase(pcl),al ;release the libraries
CALLEXEC CloseLibrary
move.1 GfxBase(pcl),a0
CALIEXEC ExitLibrary
rts               ;and exit

myinterrupt: move.l d2-d7/a2-a6/a6,-(a7) ;save registers
movev sprite     ;move the sprite
ret0007 registers
lea the next server
via the chain two

movev sprite: comp.b $0,alldone ;bounce 10 times?
beq.a movevpr007exit ;yes as skip
subi.b $1,apritedelay ;moved 160 pixels?
beq.a $160,apritedelay ;no, keep moving
subi.b $1,alldone     ;bounced once more
beq.a $16,apritedelay ;change direction

movevpr007exit: move.1 viewport(p0),a0 ;move the sprite
lea mysprite(p0),a1
move.1 16,apritedelay ;up/down 1 pixel
move.1 d3,apritemyint: lea mysprite(p0),a0 ;get a simple sprite
moveq $-1,d0         ;any sprite
CALLDEF GetSprite
comp.l $-1,d0 ;some free?
beq.a intvpr007exit ;job well, exit

move.w $10,spritedelay_height ;height is 10 pixels
lea spriteimage,a0 ;point to sprare picture
lea mysprite(p0),a1
CALLEXEC ChangeSprite ;change sprite shape
move.b $25,apritemyint: move.1 16,apritedelay ;and delay
bne.a movevpr007exit ;successful

lea spriteimage,pr007exit: rts}

get a hardware sprite

-was hit?

movevpr007exit: rts

get a simple sprite

movevpr007exit: rts

get a simple sprite

GetSprite
Listing 2: SetIntVecEx.asm

; Amiga Interrupts. By Ian Potts. ©1989 CoolSoft.

; Listing 2. Example of SetIntVector

; Assembled with DevPac V2.88 by Ellis Ltd.

include "read/include/"
include "exec/exec.lib-1" ;exec
include "hardware/intbits.ic" ;for int bits
include "hardware/mbits.ic" ;for audio DNA
include "hardware/custom.ic" ;for chip registers
include "exec/interrupt.ic" ;for server structure
include "misc/easystart.ic"

; custom

equ $ddf000 ;address of custom chips

move.l $ddf000,custom

CALLEXEC AllItems

; test

tst.l d8,not it?

begin
exit ;no, exit

move.l df,df,df

move.l $6000-1,d1 ;copy waveform into

move.l waveform[pcl],al ;memory 1644 times

soundinterrupt: cmp.w 0,81811

beq.a exit

move.l d0,waveform

move.w $6000-1,d1

move.w waveform,custom+audio

move.w $6000-1,custanaudio+acptr

la wavetable (pc),a0

clr.l d0

copywave: lea wavetable (pc),a8
copywavel: move.l (48,d8),[a1]+
addq.l $4,d8

move.l (48,d8),d8

move.w DRAF_ADDO,d0

dw .
dc.l AIMAG_ADDO

dc.l 0

move.l #soundinterrupt,soundint+IS_CODE

move.l #channel1data,soundint+IS_DATA
cal.1 soundint+VI

lea soundint [pc],al

move.l #channeldata,channeldata+soundint+IS_DATA

cal.1 channeldata,channeldata+soundint+IS_CODE
cal.1 channeldata,channeldata+soundint+IS_DATA

cal.1 soundint+VI

lea channeldata [pc],al

move.l #channeldata,channeldata+soundint+IS_CODE

CALLEXEC SetIntVector ;set up our interrupt routine

move.l db,oldint ;returned previous vector

move.l waveform,custom+audio+ac_ptr

move.w $66,custom+audio+ac_ptr

move.w $248,custom+audio+ac_ptr

move.w $1,channeldata+4 ;play sound once

lea channeldata (pc),al

brc stopsound ;start channel 0

soundinterrupt: move.w 0,81811 ;wait for sound to finish

bnc.a main

move.l oldint[pcl],al

CALLEXEC SetIntVector ;restore old interrupt code

move.l $6000-1,custom+audio+ac_ptr

move.l waveform[pcl],al

CALLEXEC FreeMem

snooze: move.w 6,custom+intena

rts

bne.a snooze

navel oldintlpchal

n ovel 	 1111TB ADDO,dO

CALLEEEC SetlntVector ;restore old interrupt code

move.l Iwavelength,d0 ;Free up sound waveform

n ovel 	 waveforelpc} al

CALLEXEC Free Hem

snooze: ;atop audio DNA

; disable audio interrupt

stopsound: move.w 6,custom+intena

move.w 6,custom+audio+acptr

rts
Fractals

The primary interest for this project was in generating fractal and pseudo-fractal images as demonstrated by MandelBrot in 'The Fractal Geometry of Nature', but as I investigated the concepts underlying his work I soon established that only the simplest forms were capable of being implemented on the hardware I had available.

What follows is a discussion of the development of a landscape modelling program that was submitted for final year B.Sc. Honours Degree project in March 1989. It has recently been converted from its original IBM PC format to run on the Amiga and has been somewhat redesigned to take advantage of the extra facilities and processing power available.

The article is in two parts, the first being a description of the techniques, algorithms, and principles behind the program and the second a guide to using it and creating your own landscapes. They have undergone only minor alterations to correctly reflect the Amiga version.

Editors note: the source code for AMOI runs to over 2000 lines and would take up a large proportion of this issue! Therefore, I'm afraid that we have had to leave it out. However, the full source code and the executable version of AMOI, are contained on the TransAmi disk for this issue.

Fractals

The primary interest for this project was in generating fractal and pseudo-fractal images as demonstrated by MandelBrot in 'The Fractal Geometry of Nature', but as I investigated the concepts underlying his work I soon established that only the simplest forms were capable of being implemented on the hardware I had available.

The most basic fractal algorithm is that of 'random mid-point displacement'. This assumes a plane to be composed of a series of smaller planes and these smaller planes to be similarly composed. The random generation of a plane of this type involves taking the four vertices of the plane and bisecting each; then moving each of these midpoints a random distance and a random direction away from their original position. The same process is performed on the centre point of the plane.

When the resulting points are connected they form a grid of four smaller planes. The process is then repeated on each of the four planes and then recursively onward until a set limiting size is reached. This concept is illustrated in figure 1.

The technique is simple to implement in any language that supports recursive procedures, and is reasonably quick to execute; there are many programs in the public domain that use this algorithm. The process suffers from several drawbacks however. The process is entirely random so the user has no control over the shape of the landscape produced; and the landscapes generated are characterised by a very jagged and rocky appearance that, though impressive, only simulates a small section of the varied terrain given to us by millennia of erosion. In all of the work by MandelBrot, I have never seen a rolling landscape.

Generating real landscapes

So, if fractal techniques offer little opportunity for realistic landscape modelling, why don't we use data provided by real life and utilise genuine Ordnance Survey data for generating our landscapes?

Well, we could do; the Ordnance Survey (o.s.) organisation have begun the huge task of computerising their maps and we had available two files from o.s. The first was a simple height grid for an area of South Wales and the second a unique attempt to represent all the information held on a normal o.s. map-sheet this time relating to areas of Yorkshire. However, the Yorkshire map alone required something in the region of 400 Mbyte of our Polytechnic's main VAX disk storage and came with a 200 page record format manual - not many people were going to type that in, and anyway the data they hold is fixed; they produce the same landscape each time and do not allow the user to express their creativity, which is of course, half the interest.

A compromise had to be found. It was decided to allowing the user to specify contour lines (using a simple mouse driven interface) and then develop code to process these contours to produce a three dimensional projection of the landscape suggested by those contours, with additional fractal
Mid-point displacement produces 4 surfaces from ABCD:

A

A

A

A

AEAD  CCEB

B

B

B

B

BBEA  DDEC

Figure 1

"influences" in an attempt to make the landscape more realistic. Further, an algorithm was defined to provide the landscape generator with some rules for rendering slopes and plains. Low-lying ground is drawn as sand gradually changing to a band of trees before fading out to rocky mountain slopes.

Aside from the basic function, the options were added to provide storage and retrieval of map information in three formats.

1 As a contour map. This can then be retrieved for amendment or for landscape reproduction at a later date (perhaps after the projection algorithms are redesigned, allowing for experimentation). Being an Amiga program, we decided on an IFF format for the contour map. This allows people with Deluxe Paint and the like to modify maps with significantly more powerful tools than those provided in AMOI.

2 As a processed grid. This is the actual calculated spatial image of the terrain. Storing a landscape in this format allows landscapes to be drawn without the need to recalculate the grid information from the contours.

3 As an IFF format image file from the final landscape projection. This provides the user with a great deal of flexibility, as well as allowing us the means to off-load the responsibility of difficult things like printing to other utilities.

The tools provided for creating contour maps are all available from the main program screen where a smallish (256x200) drawing area is bounded by a white box. Contour lines are traced in this box using the mouse and the user is provided with 15 colours to flood the desired areas (indicating heights).

A full description of the user interface is given later in the article. First, we shall describe the techniques used to take a sketched contour map to the final projected image.

Interpolation techniques

The heart of the software lies in the routines that convert the colour contour map drawn by the user. The output from the interpolation phase is a sixty-four by sixty-four grid of altitudes. The rows and columns of the grid relate to the x and y positions on the contour map.

The highest level input into the interpolation algorithm is the colour contour map drawn by the user. The output from the interpolation phase is a sixty-four by sixty-four grid of altitudes. The rows and columns of the grid relate to the x and y positions on the contour map.

The first job that the routine must perform is the elimination of contour lines. This is because the routine that gets the contour data uses the colour of a screen pixel to produce a height. Since contours are all displayed in colour fifteen (white), the
existence of contours on the map would produce spurious heights of value fifteen. To remove the contours requires a simple scanning algorithm.

Basically, the program looks at each pixel in the drawing area and establishes if it is of colour fifteen. If this is the case then the algorithm enters its search and replace mode. It will first look North of its current position by one pixel and if it finds a valid height colour, will replace the original pixel with that colour. If a non-valid colour is found then the process is repeated to the West, South, and East. If no valid colour is found in any of the four directions then the process is repeated, but this time looking two pixels away. This incrementing of the search distance is repeated until a valid colour is found.

In practice, since colour lines are rarely above one or two pixels wide there a valid colour is usually found on the first or second loop around the cardinal points. An exception to this is where a large amount of amending has been performed on a particular area of the screen. In this case, a greater number of iterations may be required; the accuracy of this algorithm deteriorates as the distance from the primary point increases.

The next phase is in many ways the most important, for it is here that the steps in the data are removed. The basic rules followed by the routine are as follows:

Any point in an area of one particular height is likely to be at an altitude between the low point of this region and the low point of the next highest region. Obviously this height would not be the average of the two region heights, but would instead be in the ratio. Put simply, the height of a point in a particular region is governed by its proximity to adjacent regions.

So an obvious implementation of this theory could examine the points bordering each point in the grid and weight its altitude by their heights. For example, if a point of altitude five were bordered by three points of altitude two, two of altitude six and three of altitude five then, since it has three adjacent points of value six and only two of value four, we could assume that it lies three fifths of the distance between the values of five and six - for example $(3/(2+3))$. So we would estimate its height to be $(5 + (3/5))$ or 5.6. If we repeat this calculation on each point of the grid we should arrive at a rounded version of the map.

There are drawbacks to this approach. Unless the contour lines are very tightly packed it is unlikely that many points will have adjacent points of different altitudes and only those lying near contour lines will benefit from the rounding technique. There are two obvious solutions to this problem.

The first would be to scan out from the point until contours of lower and greater height area are met (in a similar manner to the contour removal technique described above) and then use a similar triangles calculation to evaluate the real height of the point. The second solution would be to allow the point of interest to be modified by all the points in a certain area around it.

The core of this second solution is to generate a second grid which has, for each point, the total of all the heights in a square around that point, then to divide the total by the number of points accumulated and use this mean value as the new point height. A problem occurs with points near the edge of the grid as we cannot get values beyond the grid and so if we skip those values we get a falsely lower height because the total will still be divided by the total number of points in a normal square. The solution to this is to build a aggregate array which holds, for each point, the number of points which have affected it - then when the averaging stage is performed the total is divided by the aggregate and not the standard square total. I have coded both of these algorithms and they produce quite realistic interpolations of the contour data.

The next stage of the system is concerned with how to display the grid of points on a two dimensional display.

Fractalising, projection and shading

The three-dimensional projection algorithm is a fairly simple process that simply scales the size of the quadrilaterals that make up the picture according to their distance from the viewer. Changing the rate at which the size increases and decreases affects the apparent viewpoint of the user. Earlier versions of the software had a much higher viewpoint that seemed to be further from the landscape. The final version uses a viewpoint that gives the user the impression of being 'in' the terrain.

The first thing the landscape generator does is to put down a blue 'sea-level' plane which is a simple projection of the four corner points at zero elevation. It then performs a traversal from back to front of the $64 \times 64$ height grid. At each grid point $(i,j)$ a plane is made from the grid points $(i,j), (i+1,j), (i+1,j+1)$, and $(i,j+1)$ and the main projection function $\text{draw} \_\text{poly}()$ is called. The function takes as parameters the grid coordinates and calculated heights for each of the corner points of the plane.

$\text{draw} \_\text{poly}()$ is a recursive routine based around MandelBrot's midpoint displacement algorithm described above. Initially it projects two of the plane corners, remapping them to Amiga pixel $x,y$ coordinates, and runs a test (with a little random weighting) to determine if the plane has reached its cut-off size. The plane cut-off point should ideally be when it is of a detail that is reaching the resolution limits of the screen to such a degree that further examination of this plane would be unproductive. Experimentation has shown that a plane of less than 6 pixels wide by 5 pixels high does not gain more 'reality' by being subjected to further fractalisation.

If this is true for the plane, the corner coordinates are passed on to the rendering phase. Otherwise, $\text{draw} \_\text{poly}()$ generates a height displacement (based on the initial height) for the surface mid points and calls itself four times, once for each of the newly created surfaces. The mid-point displacements can be identified by the $\text{nz}$ prefix in the listing.
When a surface is finally accepted, it goes through two algorithms. The first is an attempt at increasing the efficiency of the projection routines and is a technique known as 'backplane removal'. This simply determines whether a plane is facing away towards the user, selecting only the facing planes for display. Although the principle is simple, it is rather mathematical.

To determine if we are dealing with a back-plane it is necessary to evaluate the surface normal of the plane and the vector of the view plane normal. If the dot product of these two vectors is negative then the plane is a back-plane and need not be drawn. For this algorithm to work it is essential that the vertices of each polygon comprising the shape are ordered in an anticlockwise manner (assuming a right-handed coordinate system).

Surface shading

When the projection algorithms have decided they have a plane that is worth rendering, their next step is to calculate the intensity of the light as it reflects from the surface of the plane. The intensity depends on four factors:

\[ I = I_a \cdot K_a + I_p \cdot K_d \cdot (L \cdot N) \]

L.N is the dot product of the normalised surface normal vector and the normalised vector of the point light source. The result of the equation is the intensity of light reflected by the plane and is used as a colour modifier during rendering to give the shade of the plane. It is necessary to specify cut-off points for \( I \) so that it stays within the bounds of the colour table.

In version 2.7 of AMOI, the vector of the point light source is a constant (sitting somewhere behind and to the left of the landscape) defined by the three values \( \mathbf{x}_l, \mathbf{y}_l \) and \( \mathbf{z}_l \). If you wish, you could alter them to move the light source. Note that \( \mathbf{x}_l, \mathbf{y}_l \) and \( \mathbf{z}_l \) are assumed to form a normalised vector. For those who don't fully understand this implies that you can't just go altering \( \mathbf{x}_l, \mathbf{y}_l \), and \( \mathbf{z}_l \) at will.

The function (called `plane.colour32()`) in the source code is reasonably incomprehensible but if you do wish to delve into it you might like note that to calculate a surface normal, it is only required that two of the plane vectors are known. These are represented by the vectors \((x_0, y_0, z_0)\) and \((x_1, y_1, z_1)\), and \((x_2, y_2, z_2)\).

The net effect of all these routines is, occasionally, a quite stunning landscape projection and the mix of sand, trees and mountain ranges can have you reaching for your suntan lotion and Bounty bar before you know what's come over you.

Of course, the first thing you'll want to do is flick on your Xerox 4020 and print out a few postcards bearing the typical smug "The weather's here, wish you were fine" kind of message so you can send them to your friends. Well, to cater for just this occasion, AMOI provides an IFF save facility.

IFF support

As explained in the initial few paragraphs, providing IFF manipulation greatly increases the flexibility of AMOI. Unfortunately, this does rather mean we have to write a whole load of support code to provide IFF read and write functions. Fortunately, a lot of the work has already been done for us by various kind people.

AMOI uses the Amiga `BitMap` structure on which to base its read and write operations, for within the `BitMap` we can find almost everything we need to know about the incoming or outgoing picture.

When loading and saving a landscape picture, the routines are called using the display screen's `BitMap` structure and so operations are performed directly to the display. Contour information though, is treated somewhat differently as the contour map is contained within a 256x200 grid on a 320x256 screen. Quite simply, a buffering `BitMap` is maintained to contain the contour information which is in due course, Blit'ed to and from the screen `BitMap`.

Although IFF is discussed in many places (most noticeably in the Appendices of the Amiga ROM Kernel Reference Manual - Exec) it is a significant portion of the listing and the routines there could very well be useful at a later date to any programmer looking to use IFF files (even if its just for loading a pre-drawn front end). To understand the IFF support, a definition of IFF is needed.

IFF and the IFF ILBM

The IFF file format was devised by Commodore and Electronic Arts, and is an attempt to provide a means of storing data in a machine independent form. This does not only apply to graphics but also to sound and animation data as well as many others. The format has quickly become the standard for file formats on the Amiga and is rapidly gaining favour in other environments.

The format is based around the concept of independent chunks, each chunk has a standard header of eight bytes that contain the name of the chunk (usually four alphanumeric characters like BODY, VHDR, ILBM) and its length. By specifying the length a program can skip over an unknown chunk by reading (and ignoring) length number of bytes and therefore arrive at the start of the next chunk.
With the majority of IFF files, the first chunk can be relied on to be a FORM type. More complex IFF files can be built on CAT and LIST types, but they don't often occur - you can cater for them if you like, but AMOI is long enough as it is! So we will have:

4 Bytes 'FORM' (identifying this as an IFF file)
4 Bytes Form length (how long the file is)
4 Bytes 'ILBM' (what sort of IFF file)

There now follows a series of chunks which are logically independent components of the form definition. A chunk is supposedly position independent within the file, for example the BMHD chunk in an IFF file may come before or after the BODY chunk, but in some cases this can be impractical.

BMHD Chunk:
4 Bytes 'BMHD' (BitMap HeaDer chunk)
4 Bytes Chunk length
20 Bytes Bitmap header data

The Bitmap Header chunk gives basic information about the format of the picture such as x size, y size, number of bitplanes, compression technique used and so on. The full details can be seen in the LoadIFF() function in the source code.

CMAP Chunk:
4 Bytes 'CMAP' (Colour MAP chunk)
4 Bytes Chunk length
n Bytes Colour information

The CMAP is a variable length chunk in that it contains 3 bytes for every colour of the image. These bytes represent the Red, Green, and Blue components of the colour and the whole range of 0-255 is employed.

In case you are wondering why 0-255 is used when the Amiga can only display 0-15 intensities of each colour then you should remember that IFF is not just an Amiga standard. It may be that the picture is to be displayed on a system that has either a greater or lesser range than the current host machine's design, so the IFF format has to rescale the 0-255 range for the host machine. In this way, no colour resolution is lost.

BODY Chunk:
4 Bytes 'BODY' (Image data)
4 Bytes Chunk length
n Bytes Interleaved bitmap

As the name suggests, the BODY chunk contains all the actual image data for the picture. However, the picture is not stored as one might immediately imagine. The basic principle is that for each complete line of the image there will be one corresponding line in each of the bitplanes of that image. This is where the phrase 'interleaved bitmap' comes in and an IFF ILM file will contain the first row from bitplane 0, bitplane 1, bitplane 2 and so on, then the second row from bitplane 0, bitplane 1 and so on. In this way the whole image is built up. This format is particularly suited to the Amiga display system.

ILBM compression

Graphical information is generally very demanding where it concerns storage. Fortunately it is also generally very receptive to compression techniques and large areas of single colours (for example, the black backgrounds in many pictures) may be represented by just a few bytes. The IFF FORM ILBM defines a compression technique known as 'Byte Run encoding' which is applied to each row of each bitplane as they are 'interleaved' together.

When data is being compressed it is examined to see if a sequence of repeating bytes exists. If a sequence (or run) is noticed, the compression routine puts down a control byte that indicates the length of the run and follows it with the byte to be repeated. The breaks in between runs (where no particular sequence worthy of special treatment can be found) are marked with a different control code, this time indicating the length of the non-run. To distinguish between a run control and a non-run control, the sign of the byte is altered.

0 to 127 indicates a run of 1 to 128 bytes.
-1 to -127 indicates a non-run of 2 to 128 bytes.

General program comments

AMOI centres around the familiar wait on an IDCMP message port. When a user action is reported, the program executes a large switch statement selecting which type of input was made.

Menu selections are usually a simple case of calling the relevant functions within the program. Gadget selection, for the Colour Ranges (defined as one big gadget) and the Draw/Fill gadget is similarly handled. Which colour was selected is calculated by processing the mouse y position, removing the need for 14 more gadgets (one for each colour).

The remaining action is the user pressing the left mouse button which toggles the mouse trail. If the user has trails a contour line outside the sketching area, all drawing is suspended until the pointer re-enters the box. As the drawing is performed by plotting a line from the last known x,y position of the mouse to the current location, a flag (FLOST) is maintained to stop drawing when the mouse is being dragged in from outside the drawing area (which could potentially produce a messy line half in and half out of the box) and vice versa. Although this isn't quite up to a Cohen-Sutherland class clipping algorithm, it's a damn sight shorter and works ok (but be carefully of creating little gaps when leaving or entering the drawing area - the colour flood will leak out of these).

Using FastFlood()

If you read the FastFlood() article in Volume 2 Issue 1 of TransAmi, you will know all about the fast replacement for the inbuilt Flood() function which has now been incorporated into AMOI (well, AMOI was the original inspiration for its development). The listing of AMOI assumes you are using the
fastflood.h file and because of this has a short sequence to ensure that AMOI has been run with a sufficient stack allocation.

As the listing mentions, I think there should be a good way of finding out the process's stack allocation, I just don’t know what it is. The method I used was to traverse several DOS structures to find the default stack for this CLI. You may remember FastFlood() requires a lot of stack space and so the code won’t let the user continue until it detects a stack of 40,000 bytes or more. If you do use FastFlood() remember that you will have to link in the fastpixel.o object from the assembler support code.

If you don’t have the FastFlood() code (or you simply like sitting around staring at your Amiga’s screen) then you can remove the #include "fastflood.h" line and replace the one call to FastFlood() with the normal Flood() call. If you do this you can probably reduce the stack limit significantly but remember that the fractal routines are highly recursive and require a fair bit of stack space themselves.

A note on compilation

When compiling AMOI, use the Motorola Fast Floating Point libraries please! The generation routines involve a lot of complex mathematics, and without FFP you may be waiting quite a while for your landscapes. If anyone wants to give me a 68881, I’ll be quite happy to tell them what sort of improvement it makes!

The AMOI user instructions

The current version (V2.7) of AMOI is designed to run on a basic 512 Kbyte Amiga, and the results can be printed (although AMOI doesn’t have any direct printing capabilities).

The generation of a landscape takes place in three distinct phases. These are:

- Creation of a contour map.
- Conversion of contour map into grid form.
- Projection of grid data into a two-dimensional view.

The second two phases are entirely performed by the computer, they only require the user to set them off and no further intervention is required. The first phase (creation of a contour map) is primarily a user process, and it is in this stage that the creativity of the user is brought to bear.

The AMOI Screen

When AMOI is run, a custom screen is opened with some simple gadgetry dotted around inside it. The main part of the screen is taken up with an empty blue box with a white border - this is the drawing area in which you will sketch the contours. Down the right hand side there will be a selection of colours (fifteen in all) and these are contour height indicators, much like a key on a normal geographical map.

Finally, somewhere across the bottom of the display there will be a small toggle switch. This should currently be displaying Draw which is telling you that if you were to click inside the drawing area, the mouse would begin to trail a thin white contour line.

The Draw switch itself can be selected to enter Fill mode. When Fill is active, clicking inside the drawing area will flood any shape defined by the mouse trails in the current contour colour. Note that selecting a contour colour assumes that the next thing you are likely to want to do is fill, and so it automatically selects fill mode for you.

Sketching contours

Sketching a contour map involves filling the entire drawing area with colour. Each of the colours represents a different height above sea level. The colours range from blue which represents sea to pink which represents the highest altitude available.

The first stage of creating a contour map is to draw the boundaries of the different altitudes, these boundaries correspond to contours in a normal Ordnance Survey Map. To draw a contour line, move the mouse pointer to the drawing box. Initially the mouse trail is turned off, so move the pointer to the start of your first contour line and then click the left button. Now when you move the mouse, it leaves a contour trail behind it. When you have completed a contour, click the left mouse button to terminate the trailing and then repeat this process until all your contours are drawn.

Once all the contours have been drawn in it is necessary to indicate the altitudes of the various regions defined by them. To do this click the left mouse button in the Draw/Fill gadget. Now when you enter the drawing area and click the left mouse button, the region you are pointing at will be filled with the colour relating to the current altitude. The current altitude is indicated by the white arrow pointing at the relevant place in the colour selection area. The altitude can be changed by clicking in any of the coloured boxes.

If you accidentally use the wrong colour you can just select the correct one and then re-fill that area. There is no limit to the amount of times you can swap between drawing and filling, so you can slowly build a complex map by completing small areas at a time.

When you have finished editing your contour map it is advisable to store it to disk for future use.

AMOI menus

The AMOI screen has a small menu bar divided into two areas; Project and Generate.

The Project menu provides access to all disk I/O (such as the saving of contours mentioned above) and the ubiquitous About and Quit options.
The first stage of the Process option is contour removal and this is pretty quick. The second, interpolation phase, depends in the number and complexity of contours you have drawn, so small islands take less time to process than complex rolling hills and valleys.

The Interpolation bar is then replaced by another eroding bar with the title 'Calculating' (which stands for 'Calculating Aggregates'). Once this bar has decayed (and it fairly whizzes by on the Amiga) the map has been generated.

You can now save this generated map to disk to eliminate the need to re-process the contours each time you wish to view the landscape. This is done with the Save Map option from the Project menu. The map data will be saved as a .MAP file. Once the file has been saved, you can now generate a view of the landscape.

To view the landscape, select the in 32 colours option hanging off the View option in the Generate menu and after a while (and one Decaying Alert) the projected landscape will appear.

You cannot view a freshly drawn landscape without processing it and so replying 'No' to the 'Shall we Process?' question will simply return you to the normal operating mode.

The View screen

When it is generating a landscape, AMOI keeps the drawing screen to the back of the display for optimal performance. When the generation is complete, the screen is brought forwards and a new menu becomes available. The viewing menu has just two options, Title Bar and Exit, which are both pretty self-explanatory. To help maintain your current screens, AMOI automatically asserts the View Screen's title bar when you exit the display mode - this ensures you don't ever lose other screens behind it.

Other options

Other facilities provided on the main screen menu bar are pretty self-explanatory. To help maintain your current screens, AMOI automatically asserts the View Screen's title bar when you exit the display mode - this ensures you don't ever lose other screens behind it.

Clear (from the Project menu) will clear the drawing area, restoring it to the blue of sea level. You are asked for confirmation.

Generate is rather more interesting in that it is used to initiate the processing of the contour information.

Now that the system has the contour details for your landscape it needs to smooth out the steps in altitude that are inherent in a contour map. First, select the Process option from the Generate menu. If you haven't saved the contour map you will be asked if you wish to do so. The screen will then be overlaid with a box containing a long horizontal bar. This is known, for no particular reason other than it sounds good, as a Decaying Alert indicating the progress of the operation.

The first stage of the Process option is contour removal and this is pretty quick. The second, interpolation phase, depends in the number and complexity of contours you have drawn, so small islands take less time to process than complex rolling hills and valleys.

Fractals (from the Generate menu) is an on/off option that allows you to select or deselect the 'fractalisation' phase of landscape generation. With experimentation, this can give a pretty good insight to the effect of the fractal algorithms and will significantly speed up the production of the landscape.

in Glorious HAM (from the View option in the Generate menu) provides for landscape generation using a HAM display (for many more colours). This option is not implemented in this version of AMOI because the HAM generation code is quite a significant project in itself (which may form the basis of a future article). It might be worth mentioning that the HAM routines also require the fastpixel.o support functions (with a minor modification).

Conclusion

The Amiga version of AMOI was written for Lattice C version 5.0 and its source currently occupies some 61 Kbyte of my hard disk. That's quite a bit of typing which we don't really expect many people are going to want to do - and that's why it is provided on the TransAmi disk for this issue.

What we do hope is that some of the techniques demonstrated in the project (such as IFF manipulation, menu control, 3D projection, backplane removal, surface shading, fractalising, and interpolation) will be useful. It's quite surprising how often the surface shading algorithms can be used, for example to make a dull 3D graph into a dynamically shaded portrayal worthy of 'The Money Program'.

If you do become fascinated with creating your own landscapes, as we did, you may be interested to know that version 3 will soon be available from us which, apart from having a much prettier front end, thanks to Deluxe Paint, implements both HAM landscapes and a user definable light source.

Transactor has kindly allowed us to use this article to advertise version 3.0 which will cost £4.99 + disk. For this you get the C and Assembler source, object (compiled with the debug option enabled to allow closer analysis), binary and several sample landscapes.

I'm afraid we don't have the resources to stockpile disks, so in order to provide a quick turnaround, we decided to quote a lower price and ask you to supply your own disk (which will be the one you get back!). It would also be a great help if you could preformat them (OFS or FFS).

Please make cheques payable to Danny Ross and send them to:

54 Fore Street,
Bradninch,
EXETER,
Devon EX5 4NN
England
I’ve been watching with some amusement a hot debate raging in an overseas User Group magazine. The subject of the debate translates to “Is a switch from C64/128 to Amiga a defection or an upgrade?” That’s not as funny as it sounds, because there are many people who like and use the 8-bit machines, and who regard them as more than adequate for their needs. For my part, I have many tasks still mounted on my C128, which I have no intention of changing. They work fine, have few if any bugs, and do their job. Superbase and Superscript files will continue in use for quite a while yet.

But I have transferred some Superbase files to my Amiga—not so much to use, but as a learning tool for Superbase Personal 2 and Superbase Professional. I reckon that there’s nothing quite the same as learning on real data. Besides, it gives me a chance to study the relative qualities of the programs on different machines.

The computer gear available for use was my C128D, an external 1571 drive, an Amiga 500 + expansion, and an external A1010 Amiga disk drive—the one that has the handy length cable. After playing around for a while, I concluded that you really do need something close to that set-up to do a reasonable job. You could do it with less, but it would be a bit tedious.

After looking around at utility programs, I settled on THE 64 EMULATOR from Readysoft. I really didn’t give a darn about emulating a C64, but I did want the “connectivity” solution it offered. The package has two parts—the software and a cable to connect the Amiga to a 1541/1571/1581 drive. The software is copyright, but the cable you can make yourself. A diagram for A500/2000 models is included with this article. No, sorry, it was drawn on a Mac!

OK, let’s start. I will detail how to shift a Superbase file from a C64/128 to the Amiga. For these notes, I will assume you want a carbon copy of the original file. So the first thing to do is to set up an Amiga file as an exact match of your C64/128 file—with one exception. Set up DATE fields as 6-figure numeric fields for now. I’ll explain why later.

Back to the C64/128. My first action was to make a backup of my data disk, with a check to ensure that there was enough room to take an export file. For most of you, this will be a block (sector) for every 2 records. If you have large records, then allow one block per record.

My preference was to EXPORT everything, even if a field or two could have been left off had I chosen. After transfer, I would have all the data on an Amiga disk, so I could fiddle around a bit if I wanted to.

I used EXPORT like this—EXPORT “filename”, “[”]. The last bit sets the left and right square brackets as dividers between information. I avoided slashes, commas and other marks as being too common in my files, choosing the square brackets because I never use them. Select your own dividers if you like, but be sure you choose something NOT in any of your data fields.

So away it went. Coffee time, while all the data got written to the disk in my 1571 drive.

With everything switched off, and a blank (initialized) Amiga disk ready, I hooked up the Amiga to the 1571, which I disconnected from the C128D. I set the 1571 as device 8. Then I turned on the Amiga, and loaded the Emulator. A small point here. I think the emulator will only access 1 side of the 1571—something to bear in mind.

The next few choices were easy. Select file copy, C64-Unit 8 to Amiga DFI: as source and destination, and finish off by clicking on the file display panel to choose the file AND Petsci->ASCII translation. Say GO, and time to watch the TV news. Well, so I thought. Nearly 400 records were moved over in less than 4 minutes (not brilliant, but adequate), so I didn’t even get to ogle the weather girl.

Still at the Amy, it was time to kick Superbase into life. What follows is OK for either Personal or Pro.

From the SET menu, I specified the field and record separators. These must match the EXPORT characters. If you have followed my [] example, you need to tell SB to use 123 and 125 ASCII.

I now selected PROCESS IMPORT. This read in the file from the Ami disk, and stowed the fields away to each record in correct sequence. Nothing too difficult here—the manual was easy to follow. I elected to haul in all data, which I could later dispose of if I found it of no use. And that, as they say, was that. Transfer all done in less than 10 minutes.

Back to the date problem alluded to above. Dates in 8-bit SB are stored as Julian dates from 1 January 1900. Dates in the Amiga versions are stored as Julian dates from 1 January 0001. So it is necessary to add DAYS (“30 Dec 1899”) or 693583 days to any date exported from an 8-bit computer to the Amiga. This is why I suggested you nominate date fields as numerics when setting up your new Amiga file.

The easiest way to do this is to use UPDATE—pick a date field and add the days to each record. Do this for each date field. Then go to EDIT-FIELD to amend the resulting numeric field(s) to date fields. Check your results by setting the date format to dd-mmm-yyyy.

But why did Precision choose 0001? Was there no 0000? Or is that my computer logic messing me up!
# Disks from Transactor for the AMIGA

<table>
<thead>
<tr>
<th>TransAmi Disk #1</th>
<th>TransAmi Disk #2</th>
<th>TransAmi Disk #3</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLink</td>
<td>StructureBrowser1.3</td>
<td>Vol 1 Issue 1</td>
</tr>
<tr>
<td>StructureBrowser</td>
<td>DumpLo</td>
<td>StructureBrowser1.3</td>
</tr>
<tr>
<td>Uedit</td>
<td>TWM</td>
<td>ARP</td>
</tr>
<tr>
<td>PopToFront</td>
<td>TalkingTasks</td>
<td>Cycles</td>
</tr>
<tr>
<td>TrapSoap</td>
<td>PeopleBall</td>
<td>MoreRows</td>
</tr>
<tr>
<td>PopColours</td>
<td>ArtThief</td>
<td>DosPackets</td>
</tr>
<tr>
<td>XE</td>
<td>PopColours1.3</td>
<td>VirusProtect</td>
</tr>
<tr>
<td>Asm68K</td>
<td>FXE-Calculator</td>
<td>MemWatch</td>
</tr>
<tr>
<td>Csh</td>
<td>TextReader</td>
<td>JBEcho</td>
</tr>
<tr>
<td>ARC</td>
<td></td>
<td>Lists</td>
</tr>
<tr>
<td>Asm6502</td>
<td>Uedit</td>
<td>FixHunk</td>
</tr>
<tr>
<td>TextReader</td>
<td>PopToFront</td>
<td>JTCalc</td>
</tr>
<tr>
<td>FnPlot.bas</td>
<td>TalkingTasks</td>
<td>ARC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TransAmi Disk #4</th>
<th>TransAmi Disk #5</th>
<th>TransAmi Disk #6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vol 1 Issue 2</td>
<td>Volume 1 Issue 3</td>
<td>Volume 1 Issue 4</td>
</tr>
<tr>
<td>MinternTutor</td>
<td>ARP</td>
<td>RunTimeLibs</td>
</tr>
<tr>
<td>ARP</td>
<td>Bits</td>
<td>DiskSalv</td>
</tr>
<tr>
<td>Cycles (updated)</td>
<td>Access</td>
<td>t</td>
</tr>
<tr>
<td>Toy3</td>
<td>EventMaker</td>
<td>Modula2</td>
</tr>
<tr>
<td>Browser</td>
<td>DeciGel</td>
<td>CallingCLI</td>
</tr>
<tr>
<td>Random</td>
<td>Copilot</td>
<td>ChangingFonts</td>
</tr>
<tr>
<td>RobPeck</td>
<td>3DPlot</td>
<td>ShellScripts</td>
</tr>
<tr>
<td>DBufDemo</td>
<td>Chatty</td>
<td></td>
</tr>
<tr>
<td>Keep</td>
<td>TextReader</td>
<td></td>
</tr>
<tr>
<td>Update</td>
<td>ARC</td>
<td></td>
</tr>
<tr>
<td>ShowInfo</td>
<td>TextReader</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TransAmi Disk #7</th>
<th>TransAmi Disk #8</th>
<th>TransAmi Disk #9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Volume 1 Issue 5</td>
<td>Volume 1 Issue 6</td>
<td>Volume 2 Issue 1</td>
</tr>
<tr>
<td>SysCheck</td>
<td>ML—Jim Butterfield</td>
<td>ML2—Jim Butterfield</td>
</tr>
<tr>
<td>Hunker</td>
<td>EXT—Dan Schein</td>
<td>FastFlood—Danny Ross</td>
</tr>
<tr>
<td>Boot68010</td>
<td>DeviceHandlers</td>
<td>MIDI—Steve Simpson</td>
</tr>
<tr>
<td>VirusX31</td>
<td>FXE Calculator</td>
<td>TrapSnap</td>
</tr>
<tr>
<td>WPUtilities</td>
<td>TextReader</td>
<td>Set CPU</td>
</tr>
<tr>
<td>Devices</td>
<td>StructureBrowser</td>
<td>ShowBoards</td>
</tr>
<tr>
<td>Val</td>
<td>Csh V2.07</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TransAmi Disk #10</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Volume 2 Issue 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML3—Jim Butterfield</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CopyDisk—Bob Rakosky</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interrupts—Ian Potts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMOI—Ross/Reynolds</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LED—Khalid Aldoseri</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXEC-dis—Markus Wandel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Script1—Khalid Aldoseri</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ArtThief</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


MIDI - the software

First steps in programming for MIDI

by Daniel Brookshier

copyright © Daniel Lee Brookshier

In the previous issue of TransAmi, Steve Simpson described the basics of MIDI hardware and took a brief look at some elementary software. In this article, Dan goes further and describes the MIDI command set and introduces some MIDI programming techniques (in Modula 2, but adaptable to other languages).

Why would you want to write your own MIDI software? In my case, I have trouble humming a tune, let alone trying to play Beethoven or Pink Floyd on my Casio keyboard. My interest lies more in the Instant Music types of software. I dream of software that prevents the novice from playing music that sounds like the torturing of furry animals.

But I find very little time to attain such a lofty goal. So I will let you in on the secrets that I have learned about MIDI and the Amiga, in hopes that you will save the world from notes played out of key.

The program example was written in Benchmark Modula 2, but is simple enough to be converted to C without too much work. In any case, there is enough information in the text to be able to write MIDI software in languages other than C and Modula 2.

The buzzword is MIDI

MIDI has only been around since the early eighties, but now most keyboards costing more than a hundred dollars or so have MIDI connections on the back.

So, what exactly is MIDI? To start with, MIDI stands for Musical Instrument Digital Interface. MIDI allows musical instruments and computers to interface and control each other using a standard communication language. Because of the complexity of today's synthesisers, and the need to connect them to computers or special devices called sequencers, MIDI has become a necessity.

Using MIDI can also indirectly aid in the writing of music by allowing the composer to record music directly from the synthesiser keyboard. Once a section of music has been recorded, it can be replayed or modified. During a live performance, sets of instrument characteristics can be modified by the press of a button, reducing the burden on the musician, who may need to change dozens of instrument parameters while playing.

MIDI and the serial interface

On a basic MIDI instrument there are three connectors. These are five pin DIN plugs that are labelled IN, OUT and THRU.

The IN, as its name implies, receives data and would normally be connected to the OUT of another instrument (or computer) which is sending messages to the instrument.

The OUT connector is used when messages from the instrument itself are to be sent to other instruments or computers.

The THRU connector is just a pass through of the signals coming from the IN connector; it allows several instruments to be controlled by one source.

Reading and writing to a MIDI adaptor on the Amiga's serial port is straightforward. Listing 2 is a Modula 2 module containing all the procedures needed to open, close, read from and write to MIDI (listing 1 is the associated .def file).

Using MIDI is very similar to normal serial communications, as might be used with a modem. In fact the setup for MIDI is almost the same for normal RS232 transmitting eight bits of data with one stop bit and no parity. The only differences are in the ioSerFlags of the serial device and the special baud rate.

Both the SerRadBoogie and SerXDisabled flags must be set, and the baud rate set to 31.25 Kilobaud. At this high baud rate, the serial device is required to work as efficiently as possible and setting the SerRadBoogie flag forces the serial device to skip much of its internal checking code.

Once the serial device has been initialised, reading or writing MIDI codes is accomplished by sending messages to the device port.
The make-up of a MIDI message

The signals that MIDI instruments send and receive come in five formats. They all consist of at least one byte, although most are two or three bytes in length.

The first byte is the status byte which is sent to signal a new message or event. It has its first bit (the most significant) set. This can be checked by verifying that the value of the byte is greater than 127.

The status byte can be followed by one or more data bytes which have the most significant bit cleared. This means that we can tell if it is a data byte simply by checking if this bit is zero, or by verifying that the value of the entire byte is less than or equal to 127.

One way to determine what you're going to do when you receive a byte from the MIDI is to use the following structure:

```pascal
IF MIDI_In > 127 THEN
  (* Handle status byte *)
ELSE
  (* Process data byte *)
END;
```

Next, we must decode the status byte. There are actually five types of status byte, each handling a different level of control. Figure 1 gives the bit patterns of these messages, along with the formats for the data. Of these, there are two groups of messages: channel messages and system messages. To start decoding, the specific message group must first be determined.

The status word is first broken into two sections, an upper and lower half. The type and group of the command is in the upper, most significant, half. The lower half contains either the channel number or the subset of the command. Decoding these numbers can be done in several ways. The first is to mask out the stems and channel separately into two new words. In C this can be quite simple. As shown below, we simply AND a constant and the status together.

```pascal
Command = 0x0F & MIDI_In;
Channel = 0x0F & MIDI_In;
```

For Modula 2, a different approach is needed. To AND in Modula 2, we first transform the status word into a set. Next the intersection of the mask and status sets are determined and then the result is converted back to an unsigned byte.

```pascal
TYPE
  MaskSet : SET OF (0..7);
VAR
  MIDI_In : BYTE;
  Command,Channel : CARDINAL;
BEGIN
  CommandMSB = CARDINAL(SHIFT(MIDI_In,-4));
  CommandLSB = CARDINAL(MaskSet(00FH) * MaskSet(MIDI_In));
```

Now that the command has been broken into nibble-sized chunks, the process of determining the message group and the command is straightforward. The DecodeMIDICommand procedure does most of the work - it extracts the data from the status byte, then case statements are used to determine the type of commands. As can be seen, in the upper half of the status word the channel messages range between 8 and 14, while the system messages are 15. With channel messages, the lower half of the status word contains the channel number whereas for system messages the lower half of the status byte contains the actual system command.

There are four different actions to be performed, depending on the type and the group of the command. The first is that it may not be a command that is supported, in which case a procedure is called to notify the user that an error has occurred.

The next possibility is that the command will be followed by one or more data bytes. To accomplish this, the expected number of bytes is set and the data byte counter is reset. The channel number and the command are saved, to be used when all the data has been received.

The third possibility is that a command has been received that needs no data. In this case, because we do not need to wait for more data from the MIDI interface, the action needed by the command can be performed immediately by calling another procedure. This brings about the fourth possible action that may occur.

If the program does not support the function, then an alternative procedure can be called to inform the user that the function is not supported. However, caution should be used in printing such messages as some of the commands can occur as fast as they can be read and so the printing of non-supported functions should be done in a debug mode only!

For both the error and non-supported commands, a variable is set to allow the program to ignore any data that may be sent after it.

Now that the status has been decoded, the following protocol rules must be followed:

**Rule 1:** Each status byte, if it needs data, expects a specific number of data bytes. No action should be performed until all the data is received.
<table>
<thead>
<tr>
<th>Function</th>
<th>Status Word</th>
<th>Data 1</th>
<th>Data 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Channel Voice Messages</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Note OFF</td>
<td>$8c</td>
<td>1000cccc</td>
<td>0xxxxxx (key number)</td>
</tr>
<tr>
<td>Note ON</td>
<td>$9c</td>
<td>1001cccc</td>
<td>0xxxxxx (key number)</td>
</tr>
<tr>
<td>Polyphonic Key Pressure</td>
<td>$Ac</td>
<td>1010cccc</td>
<td>0xxxxxx (key number)</td>
</tr>
<tr>
<td>Controller Change</td>
<td>$Bc</td>
<td>1011cccc</td>
<td>0xxxxxx (controller number)</td>
</tr>
<tr>
<td>Program Change</td>
<td>$Cc</td>
<td>1100cccc</td>
<td>0xxxxxx (program number)</td>
</tr>
<tr>
<td>Channel Pressure</td>
<td>$Dc</td>
<td>1101cccc</td>
<td>0xxxxxx (after touch)</td>
</tr>
<tr>
<td>Pitch Bend</td>
<td>$Ec</td>
<td>1110cccc</td>
<td>0xxxxxx (Bend position LSB)</td>
</tr>
<tr>
<td><strong>Channel Mode Messages</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Local Control</td>
<td>$Bc</td>
<td>1011cccc</td>
<td>01111010</td>
</tr>
<tr>
<td>All notes OFF</td>
<td>$Bc</td>
<td>1011cccc</td>
<td>01111011</td>
</tr>
<tr>
<td>Omni mode OFF</td>
<td>$Bc</td>
<td>1011cccc</td>
<td>01111100</td>
</tr>
<tr>
<td>Omni mode ON</td>
<td>$Bc</td>
<td>1011cccc</td>
<td>01111101</td>
</tr>
<tr>
<td>Mono mode ON (Poly OFF)</td>
<td>$Bc</td>
<td>1011cccc</td>
<td>01111110</td>
</tr>
<tr>
<td>Poly mode ON (Mono OFF)</td>
<td>$Bc</td>
<td>1011cccc</td>
<td>01111111</td>
</tr>
<tr>
<td><strong>System Common Messages</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Song Position Pointer</td>
<td>$F2</td>
<td>11110010</td>
<td>0xxxxxx (song position LSB)</td>
</tr>
<tr>
<td>Song Select</td>
<td>$F3</td>
<td>11110011</td>
<td>0xxxxxx (song number)</td>
</tr>
<tr>
<td>Tune Request</td>
<td>$F6</td>
<td>11110110</td>
<td></td>
</tr>
<tr>
<td><strong>System Real-time Messages</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Midi Clock</td>
<td>$F8</td>
<td>11111000</td>
<td></td>
</tr>
<tr>
<td>Start</td>
<td>$FA</td>
<td>11111010</td>
<td></td>
</tr>
<tr>
<td>Continue</td>
<td>$FB</td>
<td>11111011</td>
<td></td>
</tr>
<tr>
<td>Stop</td>
<td>$FC</td>
<td>11111100</td>
<td></td>
</tr>
<tr>
<td>Active Sensing</td>
<td>$FE</td>
<td>11111110</td>
<td></td>
</tr>
<tr>
<td>System RESET</td>
<td>$FF</td>
<td>11111111</td>
<td></td>
</tr>
<tr>
<td><strong>System Exclusive</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>System Exclusive</td>
<td>$F0</td>
<td>11110000</td>
<td>0xxxxxx (manufacturer's ID)</td>
</tr>
<tr>
<td>END System Exclusive</td>
<td>$F7</td>
<td>11110111</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**
- c indicates MIDI channel number
- x indicates arbitrary data
- n indicates number of channels to assign voices to

**Figure 1: MIDI command summary**
Rule 2: After a status byte has been received, any number of complete data messages may be sent without receiving another status byte.

Rule 3: System messages can be received at any time, even between data bytes!

Of these, rule 2 is the most important. The best way to explain it is by example. If a key is pressed on a synthesizer keyboard, the keyboard will send a status byte and two data bytes. If two keys are pressed at the same time, then the MIDI device may send one status byte and two sets of two data bytes.

Rule 2 also causes problems with a variation in the Note On command. On some instruments, the Note Off command is not used; instead, a velocity of zero in the Note On command signifies that a note should be turned off. It is a requirement that a MIDI program should take this into account to prevent notes from being stuck in the on position.

This also means that if a key has been pressed on this type of instrument before your program has started, the program will have never seen the status byte and it will be getting data that it does not know what to do with. The MIDI program should always ignore data until a status byte is received. The only problem is that you may need to press a function key on the instrument so that a status byte will be sent after the next keyboard key is hit.

The main module ReadAllMIDI (in listing 3) reads the basic MIDI note and pitch bend commands and plays up to four channels of a triangle wave through the audio device. As each command is received the note value is converted to frequency and the octave is determined. If a channel is opened, the sound is started on that channel; if the pitch wheel is moved, the new value is added to the wavelength to play. When the MIDI instruments key is released the sound on the corresponding channel is stopped. If new MIDI channel commands are to be decoded, they only need the command value and code added to the CASE statement.

The MIDI Commands

Before going on to described the MIDI command set (a summary of which is given in Figure 1), I should warn the prospective buyer of a MIDI-controlled instrument never to buy any MIDI instrument without checking the manual for supported functions. Just because a keyboard does certain things does not mean that they can be fully controlled through MIDI. For the Casio HT-700, for instance, ninety percent of its capability is not usable through the MIDI interface. The main problem is that the ability to customise the sounds on the instrument are only possible through a wheel type device. Just to change the characteristics of one instrument could take several minutes.

This brings up the first and most important command for those looking into high performance instruments, the system exclusive. Like the others, this command starts with a status byte; next comes a data byte containing the company identification. Following this is a series of bytes containing special data for the particular instrument, and the complete command is terminated with an end of system exclusive status byte. Although there are some standard formats that are common between companies, you should refer to your manual for further information.

The most common commands are the channel messages. These use the channel identification number that is decoded from the lower half of the status byte. The number ranges between 0 and 15, meaning that up to 16 MIDI instruments can be controlled on one MIDI cable.

The channel number shows what instrument, or what part of the instrument, is sending or receiving data. The channel number should be processed, because we may receive a channel message from an instrument that we want to ignore. Remember that the MIDI allows several instruments to be connected on the same cable! On the Casio HT-700 keyboard, channel numbers are also different for the rhythm and the main keyboard.

The first of the channel messages deals with what note is to be played and how they are controlled. Note On and Note Off specify the note to play or stop. The data bytes of both commands include both the note and the velocity with which the note was struck. The note is determined by starting with the value of 60 equal to middle C; adding one or subtracting one to this number adds or subtracts a semitone. On a piano, a semitone is the distance between adjacent white and black keys, between the B and C keys, and the E and F keys.

The MIDI key value must be converted to frequency to be used by the Amiga audio device. The MIDIFreqConv procedure is used to set up a table of frequencies for each note. The equation used is quite simple, multiplying the last entry in the table by a constant - although the output is quite accurate. The MIDI key number is used as an index into the table.

The velocity of the note is the force with which the note is played. An example of this would be force used when striking a key on a piano. If the key is hit with a light touch, the sound produced is low volume. If the key is struck harder, the sound produced is louder. The amount of extra electronics needed to make a keyboard record velocity is expensive so that some keyboards are just ON/OFF switches.

Note off commands are similar to the note on. The only difference is that the velocity is now an indication of how fast the sound decays to silence. Please remember that this command may be substituted with a note on velocity value of zero.

Polyphonic key pressure is another command that may be on only the more expensive keyboards. This is an indication of just how much constant pressure is being applied when playing a specific note. Most explanations of this command
say that it is used to signify vibrato. It may also be used in the
newer wind instruments, like the new MIDI saxophone. Here,
the pressure value would show how hard you are blowing into
the instrument.

*Program change* is what is often called the patch number.
Using this command, an instrument with several pre-
programmed instruments can have the selected instrument
changed. On some keyboards, parts of the range are also used
to select the rhythm.

*Control change* is a versatile command allowing control of
several different types of parameters. Figure 2 is a list of some
of the possible values that may be modified. The selection of
the parameter is in the first data byte and the new value for the
parameter is in the second byte. Remember, not all of these
may be recognised by your instrument.

*Channel pressure* is similar to *polyphonic pressure* except that,
where *polyphonic pressure* represents the pressure applied to
each *individual* note, *channel pressure* affects all notes
currently being played.

The last channel message is *pitch bend*. This is one of the
more fun commands, and is also on most new keyboards. This
code represents the position of the pitch-bend wheel - a control
devices which is often used to change the pitch (up and down)
of a note being played.

The pitch bend wheel data starts at 8192, which means that
8192 is subtracted from the pitch bend value and then added to
the value of all the notes currently playing. For the HT-700
and many other keyboards, only the most significant byte is
used so the data starts at 64. In the program example, each
pitch is added directly to the period sent to the audio device.

The *channel mode* message is used to control the keyboard
characteristics. The *local control off* command allows the
keyboard to be used only as an output device without playing
any of its own instruments. This is useful when the keyboard
is only being used to send data to the Amiga or other MIDI
devices.

The next *channel mode* message is the *all notes off* command.
This does exactly what it says - it turns all the notes off. It is a
good command to have if some of the notes get stuck on.

The rest of the channel mode messages are concerned with
how many notes can be played at once. The *poly* commands
allow two or more notes to be played at once while the *omni*
mode allows only one note to be played at a time. If more
notes are sent than are expected, the newer notes will be
ignored.

**System Real-Time messages**

The next set of MIDI commands are the *system real time*
messages. They consist of only one byte and may appear
between other messages. This is very important to remember.
If you have been receiving note events and then get a *system
real time* message, then the next data byte should still be
handled as a note event. These messages are all associated
with sequencing and they control the starting, stopping and
timing of events.

The most important of these commands is the timing signal
which is sent every time the lights blink on the rhythm section
of most keyboards. The signal is used as a metronome. It is
best used as a constant update for timing. Because not all
instruments are alike a common timing source is needed. An
instrument with a rhythm section should have a switch to go
between sending and receiving this command.

MIDI is a very capable interface that allows a lot of flexibility.
The example in this article falls way short of using the
complete set of commands possible, but they can be added
with very little work. Just remember the rules and don’t hit
any notes out of key.

<table>
<thead>
<tr>
<th>Continuous controllers (MSB)</th>
<th>Continuous controllers (LSB)</th>
<th>Switch controllers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0  undefined</td>
<td>32  undefined</td>
<td>64  Sustain pedal</td>
</tr>
<tr>
<td>1  Modulation wheel</td>
<td>33  Modulation wheel</td>
<td>65  Portamento</td>
</tr>
<tr>
<td>2  Breath controller</td>
<td>34  Breath controller</td>
<td>66  Sostenuto</td>
</tr>
<tr>
<td>4  Foot controller</td>
<td>36  Foot controller</td>
<td>67  Soft pedal</td>
</tr>
<tr>
<td>5  Portamento time</td>
<td>37  Portamento time</td>
<td>69  Hold 2</td>
</tr>
<tr>
<td>6  Data entry MSB</td>
<td>38  Data entry LSB</td>
<td>83-95 General controllers</td>
</tr>
<tr>
<td>7  Main volume</td>
<td>39  Main volume</td>
<td></td>
</tr>
<tr>
<td>8  Balance</td>
<td>40  Balance</td>
<td></td>
</tr>
<tr>
<td>10 Pan</td>
<td>42  Pan</td>
<td></td>
</tr>
<tr>
<td>11 Expression controller</td>
<td>43  Expression controller</td>
<td></td>
</tr>
<tr>
<td>12-31 General controllers</td>
<td>44-63 General controllers</td>
<td></td>
</tr>
</tbody>
</table>

Note:
- If a continuous controller requires a data range > 0-127, two controller parameters may be used; one containing the MSB of the data and the other the LSB using different device numbers.
- For data in the range 0-127, only the MSB device numbers are used.
- Some manufacturers use non-standard controllers and therefore non-standard controller numbers.

*Figure 2: Controller parameter device numbers*
**Figure 1: MIDL.def**

```plaintext
* Name: MIDL.def
* Created: 01/08/89 Updated: 00/00/89 Author: Daniel Brockhier
* Description: MIDI Read/Write utilities

DEFINITION MODULE MIDI;

FROM SYSTEM IMPORT ADDRESS, BYTE;
FROM Ports IMPORT MagPortPtr;
FROM SerialDevice IMPORT IOExtSerPtr;

PROCEDURE SendWriteSerial(VAR io : IOExtSerPtr;
                          data : ADDRESS;
                          Length : LONGCARD) : INTEGER;

PROCEDURE StartMIDI(VAR IOser : IOExtSerPtr;
                     VAR Port : MagPortPtr) : INTEGER;

PROCEDURE ReadSerial(VAR io : IOExtSerPtr;
                     data : ADDRESS;
                     Length : INTEGER);

PROCEDURE ReadSerialByte(VAR io : IOExtSerPtr;
                          VAR ByteStorage : BYTE);

PROCEDURE WriteSerial(VAR io : IOExtSerPtr;
                       data : ADDRESS;
                       Length : LONGCARD);

PROCEDURE WriteSerialByte(VAR io : IOExtSerPtr;
                          VAR ByteStorage : BYTE);

**Figure 2: MIDL.mod**

```plaintext
* Name: MIDL.MOD
* Created: 01/08/89 Updated: 00/00/89 Author: Daniel Brockhier
* Description: MIDI Read/Write utilities

IMPLEMENTATION MODULE MIDI;

FROM PrintTeminal IMPORT printf;
FROM FormatString IMPORT Pormat, PointArg;
FROM SYSTEM IMPORT ADDR, ADDRESS,
FROM IODevices IMPORT CsrNanstd, IOStdReg, IORead, IOWrite, CloseDevice;
FROM DevicesUtil IMPORT CreateExtIO, DeleteExtIO;
FROM Portslutil IMPORT CreatePort, DeletePort;
FROM Ports IMPORT MagPortPtr;
FROM Inout IMPORT ReadInt;

NOTE: details of the SerialDevice imports are given as comments

FROM SerialDevice IMPORT

*CONSTM
SerialName ("serial.devim"),
SDConQuery (* = ConMstd + 0; *),
SDConBreak (* = ConMstd + 1; *),
SDConSetParams(* = ConMstd + 2; *),

*YFEY*
array of termination char's

IFIRsyst := (* SYSTEM TO IFIRsyst*),
IFIRsyst := (** REPROD
Termsys := LONGCARD;
TermAcryp := LONGCARD;
NO;)

*CONSTM
(* 1080Request.ioFlags := IFIRflagsSet[ ] *)
LCserFullRead (* = 7; from read buffer bit *),
LCserCmd (* = 6; rapt-quad bit *),
LCserAbort (* = 5; rapt-shorted bit *),
LCserExtio (* = 4; rapt-quad-to-current bit *),
```
**Type**

- `SezParityOn`: parity-enabled bit
- `SezParityOn`: parity feature enabled bit
- `SezWml`: (52327 bit)
- `SezQueueAck`: queue this Break (adept)
- `SezReadGrace`: high-speed mode active bit used for MIDI
- `SezHan`: non-exclusive access bit
- `SezMode`: BBE mode enabled bit
- `SezDish`: non-soft feature disabled bit

**SezSelFlage**

- `SezSelFlage`: SET OF SezSelFlage;

**SezExtFlage**

- `SezExtFlage`: SET OF SezExtMark;

**SezStatus**

- `SezStatus`: SET OF I0EStatus;

**PROCEDURE**

- `AbortIO`: VAR I0Err: I0ExtSerPtr; VAR Port: INPortPtr;

- `READSERIAL`: VAR io: I0ExtSerPtr; data: ADDRESS; length: LOWCARD; ERREAD;

**CONST**

- `SezRomBusy`: I0ExtSerPtr
- `SezReadMidmatch`: I0ExtSerPtr
- `SezReadStatus`: I0ExtSerPtr
- `SezReadError`: I0ExtSerPtr

**BEGIN**
PROCEDURE ReadSerialByte(VAR io: IOExtSerPtr; VAR ByteStorage: BYTE); VAR 
err : INTEGER; 
BEGIN 
(* Set up External I/O with parameters for a read *) 
(* Use Input port created by Start MIDI *) 
(* ByteStorage - byte to store the serial data to *)
END;

PROCEDURE WriteSerialByte(VAR io: IOExtSerPtr; VAR ByteStorage: BYTE); VAR 
err : INTEGER; 
BEGIN 
(* Set up External I/O with parameters for write *) 
(* io - external I/O port created by Start MIDI *) 
(* ByteStorage - byte to send to MIDI devices *)
END;
Create an external I/O port. This allows you to send parameters to the serial device.

```
IOPer := CreateExtIO(Port', SIZE (IOExtSer1);  
IF (IOPer = NIL) THEN  
  printf("I/O port create failed', Pargc);  
  AbortMIDI(IOPer, Port);  
  RETURN(-1);  
END;

/* Note use of SerRadBoogie flag */  
IOPer := CreateExtIO(Port', SIZE (IOExtSer1);  
PORT := NIL;  
printf("Hello Create Palled 
', Pargc);  
AbortMIDI(IOPer, Port);  
RETURN(-1);  
END;

Note use of SerRadBoogie flag! */  
IORier := SerPlagesetlkrRadlce, SerYDieabledl;  
Nana: ReadalKIOI.ad
```

This is a simple demonstration of reading a MIDI keyboard connected to the serial port. The keyboard that I use is a Casio ET-700. The MIDI interface cable is from Nineties. This example should work with any keyboard or interface. Note that all the MIDI commands and decoded, but not all have been implemented with an Amiga audio device equivalent.

```
/* The following fragment sets up all the parameters need by */  
/* the serial device to talk to MIDI devices. */  

MODULE ReadAlIMIDImod

supports modules */  
FROM MIDI IMPORT StartMIDI, AbortMIDI, ReadSerialByte, WriteSerialByte;  
FROM MIDI IMPORT UserCheckRegin, UserCheckMod;

/* Benchmark modules */  
FROM SerialDevice IMPORT WriteString;  
FROM RandomNumbers IMPORT Random;  
FROM SYSTEM IMPORT ACK, BYTE, TSIEE, ADDRHS, INRSNE, SNIFTE;  
FROM AmigaDOSProcess IMPORT Delay;  
FROM Libraries IMPORT OeaLibrary, CloseLibrary;  
FROM Libraries IMPORT IOExtSerPtr;  
FROM Libraries IMPORT ByteArray,baumix;  
FROM Libraries IMPORT AudioDev, ADCmdAllocate, AMmd]AR, IOAudio,  
ADCmdPin, ADCmdPerVol, ADCmdPerVol;  
FROM Libraries IMPORT DeleteExtIO, CreateExtIO;  
FROM Libraries IMPORT OpenDevice, CloseDevice, DevicePtr, IOPlagsSet,  
BeginIO, CBckIO, Dolt NaitID, SendIO, IOPlack,  
ReadSerialEOBCl, ADR.lblbffl), IDI;  
FOR Y := 1 TO 30 DO  
  WriteSerialEOBCl, ADR.lblbffl), IDI;
END;
```

The Serial port is now configured. Use AbortMIDI to kill.

```
FOR Y := 1 TO 30 DO  
  WriteSerialEOBCl, ADR.lblbffl), IDI;
END;
```

Wavelength of each of the wave forms for each octave.

```
Wavelength := 16;
Wavelength := 32;
Wavelength := 64;
Wavelength := 128;
```
This procedure fills an 8 bit array with a triangle wave. Random noise has been added for a more real sound. The value of Swing is used to change the direction that data is added to the wave.

PROCEDURE ChangeSound(Channel : CARDINAL; Volume : CARDINAL; Period : CARDINAL; VAR Control : IQludio);
VAR
result : INTEGER;
BEGIN
WITH Control DO
ioaRequest.ioReguest := ADDRESS(Channel);
ioaPeriod := Period;
ioaVolume := Volume;
END;
BeginIO(AADR(Control));
END ChangeSound;

PROCEDURE FillData(VAA SoundData : ADDRESS; NavePeriod : CARDINAL; Swing,( : CARDINAL; Sample, Direction : REAL; User : POINTER is APPA [0.16384] OF CHAR; Noise : IANGCPPD);
BEGIN
User := SourceData;
Direction := -127.0 / real(NavePeriod DIV 4);
Sample = 0.0;
FOR Swing = 0 TO 3 DO;
IF ((Swing - 0) OR (Swing • 1) OR ( Swing • 3 )) THEN
Direction := Direction • ( - 1. 0 );
END;
FOR I :- I(NavePeriod DIV 4) • Swing) TO ((NavePeriod DIV 4)•(Swingal)) DO
\( \text{random}(50) - 20; \) I• Creates a number from -2 to +2 9
Sample = Sample + Direction + reallNoise) ;
\( \text{Convert the sample to B bit data} \)
F Sample < 0.0 THEN
User^III := \( \text{entier(Sample)t256D} \);
ELSE
User^III := \( \text{entier(Sample)} ; \)
END;
END FillData;

This procedure stops the audio device.

PROCEDURE StopAudio4;
BEGIN
DeletePort(control.ioaReguest.ioaMessage.messagePort);
WriteString("Deleted the control port");
DeletePort(finish.ioaReguest.ioaMessage.messagePort);
WriteString("Deleted the finish port");
DeletePort(ssl.ioaReguest.ioaMessage.messagePort);
WriteString("Deleted Post 0");
DeletePort(s2.ioaReguest.ioaMessage.messagePort);
WriteString("Deleted Post 1");
WriteString("Closing the control device");
CloseDevice(control);
WriteString("Deleting the memory");
IF (control := NIL) THEN FreeMem(control, TSIZE(IOAudio)); END;
IF (control := NIL) THEN FreeMem(control, TSIZE(IOAudio)); END;
IF (control := NIL) THEN FreeMem(control, TSIZE(IOAudio)); END;
IF (control := NIL) THEN FreeMem(control, TSIZE(IOAudio)); END;
END StopAudio4;

This procedure gets a message reply port

PROCEDURE GetMessagePort(VAR replyPort : ADDRESS; Name:ADDRESS; Priority:INTEGER);
BEGIN
replyPort := CreatePort(Name,Priority);
IF replyPort = NIL THEN
WriteString("Failed to get port creation");
END;
END GetMessagePort;

This procedure, given waveform control data, modifies a channel currently playing on the given channel.

PROCEDURE StartAudioChannel(VAR Central : IOAudio; DataAddress : ADDRESS; DataLength : CARDINAL; Period : CARDINAL; Volume : CARDINAL);
BEGIN
WITH Control DO
ioaRequest.ioRequest := CommandWrite;
ioaRequest.ioTags := TOByteSet(10000000,0,IOquick);
imdata := DataAddress ;
incycles := 0 ;
inlength := DataLength ;
inPeriod := Period ;
inVolume := Volume ;
END;
BeginIO(AADR(Control));
END StartAudioChannel;

This procedure, given waveform control data, modifies a channel currently playing on the given channel.

PROCEDURE SetAudio4;
VAR
success : INTEGER;
Scale : CHAR;
BEGIN
WriteString("Entered Setup");
Scale := CHAR;
WITH Control DO
ioaRequest.ioRequest := CommandWrite;
ioaRequest.ioTags := TOByteSet(10000000,0,IOquick);
imdata := DataAddress ;
incycles := 0 ;
inlength := DataLength ;
inPeriod := Period ;
inVolume := Volume ;
END;
BeginIO(AADR(Control));
END SetAudio;

This procedure, given waveform control data, modifies a channel currently playing on the given channel.

PROCEDURE GetData(VAR SoundData : ADDRESS; WavePeriod : CARDINAL; Swing,( : CARDINAL; Sample, Direction : REAL; User : POINTER is APPA [0.16384] OF CHAR; Noise : IANGCPPD);
BEGIN
User := SourceData;
Direction := -127.0 / real(NavePeriod DIV 4);
Sample = 0.0;
FOR Swing = 0 TO 3 DO;
IF ((Swing - 0) OR (Swing • 1) OR ( Swing • 3 )) THEN
Direction := Direction • ( - 1. 0 );
END;
FOR I :- I(NavePeriod DIV 4) • Swing) TO ((NavePeriod DIV 4)•(Swingal)) DO
\( \text{random}(50) - 20; \) I• Creates a number from -2 to +2 9
Sample = Sample + Direction + reallNoise) ;
\( \text{Convert the sample to B bit data} \)
F Sample < 0.0 THEN
User^III := \( \text{entier(Sample)t256D} \);
ELSE
User^III := \( \text{entier(Sample)} ; \)
END;
END GetData;

This procedure sets up four audio channels.

PROCEDURE SetUpAudio4;
VAR
success : INTEGER;
Scale : CHAR;
BEGIN
WriteString("Entered Setup");
Scale := CHAR;
WITH Control DO
ioaRequest.ioRequest := CommandWrite;
ioaRequest.ioTags := TOByteSet(10000000,0,IOquick);
imdata := DataAddress ;
incycles := 0 ;
inlength := DataLength ;
inPeriod := Period ;
inVolume := Volume ;
END;
BeginIO(AADR(Control));
END SetUpAudio;
PROCEDURE StopAudioChannel(VAR Playing : BOOLEAN; 
VAR SoundChannelPtr : IOAudioptr;
UnitNumber : INTEGER;
VAR FinishPtr : IOAudioptr;
VAR
success : INTEGER;
BEGIN
Playing := FALSE;
success := AbortIOLSoundChannel(Playing,SoundChannelPtr,FinishPtr,UnitNumber);
success := WaitForFinsih(FinishPtr);
END StopAudioChannel;

PROCEDURE NotImplementedInSpec(Command : BYTE);
BEGIN
NotImplementedInSpec;
END NotImplementedInSpec;

PROCEDURE NotImplementedInProgram(Command : BYTE);
BEGIN
NotImplementedInProgram;
END NotImplementedInProgram;

PROCEDURE MIDI2FreqConv(VAR KeyFreq : ARRAY OF REAL);
VAR
NoteCalc : REAL;
X : CARDINAL;
BEGIN
NoteCalc := HEAL
X := CARDINAL
BEGIN
NoteCalc := log(523.3);
NoteCalc := 65.106;
FOR A = 0 TO 11 DO
KeyFreq[1] := NoteCalc;
NoteCalc := NoteCalc + 1.059163;
END;
END MIDI2FreqConv;

PROCEDURE DecodeMIDICommand(Command : BYTE);
VAR
MIDICommand : CARDINAL;
VAR PowerOfDataBytes : CARDINAL;
VAR CurrentDataCount : CARDINAL;
VAR Channel : CARDINAL;
VAR IgnoreData : BOOLEAN;
TYPE
MaskSet = SET OF (B..7);
VAR
CommandMSB : CARDINAL;(* Upper half of command word *);
CommandLSB : CARDINAL;(* Lower half of command word *);
BEGIN

This procedure is the heart of the program. Initialization routines are called then a loop is entered that reads the MIDI port one byte at a time and controls the Amiga sound in response to key and pitch bend events.

The basic logic of the program is as follows:

- Setup MIDI & Audio device
- Loop
  - Read a word from MIDI port
  - If IP > 127 THEN Determine and amount of data needed
  - ILSIP correct amount of data received
  - IF the emend ='Pitch Bend Event' THEN
    - Modify all playing channels with word2
  - ELSE the coward - 'Note Event' THEN
    - IF the datai • 'Note Off' THEN Stop the sound of the key
    - ELSE datai ='Mote On' THEN
      - Convert key to frequency to play
      - Select proper waveform for octave
      - Start the sound of the key on an open channel
  - END
- END
- END

CASE CommandSB OF
  8..11 : (* Note Off *)
    (* Note On *)
  (* Polyphonic Key Pressure *)
  (* Control/Mode Change *)
  NameOfDataBytes := 2;
  CurrentDataCount := 8;
  MIDICommand := CommandMID; %
  Channel := CommandGSB;
  END

CASE CommandSB OF
  8 : (* Timing Clock *)
    NotImplementedInProgram(Command); [
  9 : (* Start *)
    NotImplementedInProgram(Command); [
  10 : (* Continue *)
    NotImplementedInProgram(Command); [
  11 : (* System Real Time Message received. *)
    (“Process immediately. ”)
    CASE CommandSB OF
      8 : (* Programming *)
        NameOfDataBytes := 3;
        CurrentDataCount := 8;
        MIDICommand := CommandMID; %
        Channel := CommandGSB;
      9 : (* End Channel *)
        END
  END

CASE CommandLSB OF
  8..13 : (* Program Change *)
    NameOfDataBytes := 1;
    CurrentDataCount := 8;
    MIDICommand := CommandMID; %
    Channel := CommandGSB;
  14 : (* Pitch Bend Change *)
    NameOfDataBytes := 2;
    CurrentDataCount := 8;
    MIDICommand := CommandMID; %
    Channel := CommandGSB;
  15 : (* System Exclusive *)
    IF CommandLSB > 7 THEN (* System Real Time Message received. *)
      (“Process immediately. ”)
      CASE CommandLSB OF
        8 : (* Note Off *)
          (* Note On *)
        (* Polyphonic Key Pressure *)
        (* Control/Mode Change *)
        NameOfDataBytes := 2;
        CurrentDataCount := 8;
        MIDICommand := CommandMID; %
        Channel := CommandGSB;
      END
      END
  END

CASE CommandLSB OF
  8..13 : (* System Exclusive *)
    (“Not Implemented in all data is ignored “)
    NotImplementedInSpec(Command); I
    IgnoreData := TRUE;
    MIDICommand := CommandMID; %
    Channel := CommandGSB;
  14 : (* Pitch Bend Change *)
    NameOfDataBytes := 2;
    CurrentDataCount := 8;
    MIDICommand := CommandMID; %
    Channel := CommandGSB;
  15 : (* System Exclusive *)
    (“Not Implemented in all data is ignored “)
    NotImplementedInSpec(Command); I
    IgnoreData := TRUE;
    MIDICommand := CommandMID; %
    Channel := CommandGSB;
  END

CASE CommandLSB OF
  8..13 : (* System Exclusive *)
    (“Not Implemented in all data is ignored “)
    NotImplementedInSpec(Command); I
    IgnoreData := TRUE;
    MIDICommand := CommandMID; %
    Channel := CommandGSB;
  14 : (* Pitch Bend Change *)
    NameOfDataBytes := 2;
    CurrentDataCount := 8;
    MIDICommand := CommandMID; %
    Channel := CommandGSB;
  15 : (* System Exclusive *)
    (“Not Implemented in all data is ignored “)
    NotImplementedInSpec(Command); I
    IgnoreData := TRUE;
    MIDICommand := CommandMID; %
    Channel := CommandGSB;
  END

PROCEDURE TestMIDI; VAR
  Channel : CARDINAL; (* Current channel *)
  NameOfDataBytes : CARDINAL; (* Number of data bytes to wait for. “)
  IgnoreData : BOOLEAN; (* Used to ignore unwanted data. “)
  PeriodOfMIDI : CARDINAL; (* Period of MIDI pitch change “)
  Key1, Key2, Key3 : CHAR;
  ISRser : IDExtSerPtr;
  Port : MsgPortPtr
  MIDI In ; CMR;
  NotexRef : ARMY(0..120) OF REAL; (* MIDI pitch to freq conversion “)
  CurrentDataCount : CARDINAL;
  CommandBuffer : ARMY il..101 OP CHAR;
  MIDICmmd : CARDINAL;
  Kerr : INTEGER;
  ByteaReceived : CARDINAL;
  Pitchfend : CARDINAL; (* Pitch bend Modifier for all channels “)
  yReleased : CHAR; (* Frequency of key released “)
  Length0, Length1, Length2, Length3 : CARDINAL;
  Playing0, Playing1, Playing2, Playing3 : BOOLEAN;
  Period0, Period1, Period2, Period3 : CARDINAL;
  Volume0, Volume1, Volume2, Volume3 : CARDINAL;
BEGIN
  WriteString("Entered Test\n");
  IF MathTransBase = NIL THEN
    MathTransBase := OpenLibrary(\"Math\", MathTransBase, 64);
  IF MathTransBase = NIL THEN
    WriteString("Math Library failed to open \"\")
    HALT;
  END;
  END;
END;
SetUpAdNull;

SetUpAdNull;

WriteString("Exiting Setup into Tilea");

�性Data(Note0, Wavelength0);

属性Data(Note1, Wavelength1);

属性Data(Note2, Wavelength2);

属性Data(Note3, Wavelength3);

(* -------------------------- *)
(* Starting MIDI allocation. *)
(* -------------------------- *)

err := StartMIDI(IOMessagePort);
WriteString("MIDI Started");

(*--------------------------*)
(* Set up Defaults. *)
(*--------------------------*)

PitchEnd := 0; (* Default pitch bend modifier *)

Playing0 := FALSE;

Playing1 := FALSE;

Playing2 := FALSE;

Playing3 := FALSE;

Volume0 := Vol;  (* Default volume *)

Volume1 := Vol;  (* Default volume *)

Volume2 := Vol;  (* Default volume *)

Volume3 := Vol;  (* Default volume *)

BytesReceived := 0;

WriteString("MIDI Loop Entered");

(*-------------------------------*)
(* Loop until 3000 MIDI bytes or *)
(* an error occurs. *)
(*-------------------------------*)

WHILE (err = FALSE AND BytesReceived < 3000) DO

(* Make sure there are no messages waiting on the ports. *)

WHILE GetMsg(control, iOMessage, Message, ReplyPort) <> -1 DO END;

WHILE GetMsg(finish, iOMessage, Message, ReplyPort) <> -1 DO END;

WHILE (GetMsg("","\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\",\"  

IF (NumOfDataBytes = CurrentDataCount) THEN

CurrentDataCount := 3;

ELSE

INC (ByteSeceived);

END;

(* If a command was received then check the type of command. *)

IF (MIDI In > CHAR(127)) THEN (* A MIDI command has occurred. *)

DecodeMIDICommand(MIDI In,

MIDICommand,

NumOfDataBytes,

CurrentDataCount,

Channel,

IgnoreData);

ELSE

(*--------------------------*)
(* If not a command then we have data. Check for the correct *)
(* amount before allowing further processing. *)
(*--------------------------*)

IF (NumOfDataBytes = CurrentDataCount) THEN

CurrentDataCount := 3;

ELSE

INC (ByteSeceived);

END;

(*--------------------------*)
(* Calculate the octave to pick a good wave. *)
(* The lower the octave the more data we send. *)
(*--------------------------*)

CASE (CARDINAL(CommandBuffer[2]) - 34) DIV 2 OF

1 : OctTone := Note3; LengthTone := Wavelength3;

2 : OctTone := Note2; LengthTone := Wavelength2;

3 : OctTone := Note1; LengthTone := Wavelength1;

4 : OctTone := Note0; LengthTone := Wavelength0;

ELSE

END;
IF Period0 < 126 THEN
WriteString("Period is less than the minimum of 126\n");
END;
/* Start sound on open channel number. */
/*---------------------------------------------*/
IF (NOT Playing0) THEN
Playing0 := TRUE;
Key0 := CommandBuffer[1];
Period0 := Period0Value;
Oct0 := Oct0Value;
Length0 := Length0Value;
StartAudioChannel(0, Oct0, Length0, Period0 + Pitch0, Volume0);
ELSEIF (NOT Playing1) THEN
Playing1 := TRUE;
Key1 := CommandBuffer[1];
Period1 := Period1Value;
Oct1 := Oct1Value;
Length1 := Length1Value;
StartAudioChannel(1, Oct1, Length1, Period1 + Pitch1, Volume1);
ELSEIF (NOT Playing2) THEN
Playing2 := TRUE;
Key2 := CommandBuffer[1];
Period2 := Period2Value;
Oct2 := Oct2Value;
Length2 := Length2Value;
StartAudioChannel(2, Oct2, Length2, Period2 + Pitch2, Volume2);
ELSEIF (NOT Playing3) THEN
Playing3 := TRUE;
Key3 := CommandBuffer[1];
Period3 := Period3Value;
Oct3 := Oct3Value;
Length3 := Length3Value;
StartAudioChannel(3, Oct3, Length3, Period3 + Pitch3, Volume3);
END;
/*---------------------------------------------*/
/* If we got here without starting the new note it is lost */
/* If the midi events were fully buffered this would not */
/* happen. In this configuration I have lost very few. */
/*---------------------------------------------*/
END;
ELSEIF MIDICommand = 12 THEN (* MIDI Frg Change *)
/*---------------------------------------------*/
/* Change the type of instrument here. */
/*---------------------------------------------*/
END;
END;(* Data Received *)
END;(* A MIDI command has occurred. *)
END;
/*---------------------------------------------*/
/* Main loop has completed. Close audio & MIDI ports. */
/*---------------------------------------------*/
WriteString("MIDI Loop ended");
IF Playing0 THEN StopAudioChannel(Playing0, a0, u0, finish); END;
IF Playing1 THEN StopAudioChannel(Playing1, a1, u1, finish); END;
IF Playing2 THEN StopAudioChannel(Playing2, a2, u2, finish); END;
IF Playing3 THEN StopAudioChannel(Playing3, a3, u3, finish); END;
AbortMIDI([30sec, Fact]);
StopMidi();
CloseLibrary(MathTransBase);*
END TestMIDI;
Disk Copying

The subtle technique of using the trackdisk.device

by Bob Rakosky
Copyright © 1989 Bob Rakosky

Bob Rakosky is a Data Processing consultant for AGS Information Services, Inc., and is the author of the hard-disk backup utility saf-T-net (published by Metadigm), and of the Amiga version of the game Empire (published by InterStel).

The Amiga floppy disks can be accessed at a number of levels. The standard way that we read from and write to disks are by means of the FileSystem, which manages the space on the disks, and keeps track of logical groupings of data called files. It is through the file system that we can get a directory listing of the available files, find out how much space is currently unallocated on the disk, read the data that is contained in files, and create new files.

The FileSystem is device independent. It sees the data, as stored on any file system device, as a collection of blocks of data, each containing 512 bytes. All it needs to know about a device is the number of 512-byte blocks of data that are available. It reads and writes data in terms of the relative block number, starting at relative block 0 and continuing to relative block n-1, where n is the number of blocks that the device can hold.

Since my desire is to make a copy of a disk, I could accomplish this by using the FileSystem, assuming that I already had a blank, formatted disk onto which I would write all of the files found on the source disk.

But what if I didn’t have a formatted disk? What if the disk that I’m copying wasn’t created by the AmigaDos file system, and contains data not grouped into files? Even if the disk is a standard FileSystem disk, copying it file-by-file can be pretty slow, especially if the disk contains lots of small files. Copying at the device level, rather than the through the FileSystem, can be much faster as long as we are copying from and to identical devices. This is the level at which I will be accessing the floppy disks.

There is yet another level by which we could access the floppy drives, by going directly to the hardware. Since my intention is to only access standard Amiga format disks, this approach would be overkill for our needs and won’t be addressed in this article.

Data organisation

The trackdisk.device driver views the data as it is physically stored on the medium, as a series of tracks of encoded data. Each track on an Amiga disk holds 11 sectors of data (plus a little extra information not used by the file system), and each sector contains 512 bytes of data, after it is decoded. (Note that there need not be the exact correspondence between the 512-byte block size used by the file system and the 512-byte sector size of the trackdisk, but it does simplify things.)

The actual layout of the data within the track is unimportant for our purposes, as those details are handled for us by the device driver. We don’t have to know how the data is encoded, nor do we have to know how to decode the control information contained on the physical media.

A given device is configured to contain a fixed number of tracks per surface, and will contain a fixed number of surfaces. For the standard Amiga 3.5" floppy drives, each surface contains 80 tracks, and the disk contains two useable surfaces. For the standard Amiga 5.25" drive, the geometry is two surfaces of 40 tracks each.

Input and output requests are made to the trackdisk device in terms of relative byte address, starting from 0 and continuing up to the capacity of the drive (880K for the 3.5" drives). Requests must be made only to addresses that fall on a sector boundary (the byte address must be a multiple of 512), and the length of data read or written (also in bytes) must be full sector amounts (also multiples of 512). The device driver will only read or write data in terms of full tracks, however, and so must maintain an internal buffer to hold the entire track. Since we will be reading and writing entire disks, it makes more sense to perform all of our I/O operations in full track units, since that is how the data will be manipulated internally by the driver.

Using device drivers

The Amiga supports its peripheral hardware devices through software modules called device drivers. The driver for the floppy disks is contained in the Kickstart code, and is accessed in the same way that all Amiga devices are accessed.
Access to the device is achieved using the Exec library's OpenDevice() function, to which you specify the device name and the unit number. After this, input and/or output commands are sent to the device using the DoIO() or SendIO() functions, passing an IORequest block structure.

These two functions are similar, except that the DoIO() function call will not return until the actual input/output action has completed, whereas SendIO() will schedule the I/O action, but will (possibly) return before the operation has actually completed. In the latter case, your program is free to perform other processing while the actual I/O is being performed.

The communication between your task and the (separate) task that actually performs the I/O processing is accomplished by Exec's message-passing facilities (the first part of the IOREquest structure is an Exec Message structure) and, in the case of asynchronous I/O, this message is ReplyMsg()'ed by the device-driver task to be received by your task at the mn_ReplyPort (MsgPort) contained in the Message. This can then be Wait()'ed for, or tested for with GetMsg().

A full discussion of device I/O on the Amiga would take much more space than this article will allow. A full discussion, on a general level, takes up a full chapter in the ROM Kernel Manuals. We will limit our discussion to issues directly related to the accessing of floppy disks, as illustrated in the program source listing accompanying this article.

I/O with the trackdisk.device

The trackdisk.device supports three groupings of command requests: standard device commands, trackdisk-specific commands, and 'extended' trackdisk-specific commands. The accompanying program, copydisk.c, utilises commands that fall into the first two of these categories. The commands we will be discussing and using are:

- **Standard Commands:**
  - **CMD_READ** - Reads one or more sectors
  - **CMD_WRITE** - Writes one or more sectors

- **Trackdisk-specific Commands**
  - **TD_MOTOR** - Turns the motor on or off
  - **TD_FORMAT** - Initialises one or more tracks
  - **TD_GETNUMTRACKS** - find out the number of total tracks supported by the device.

As stated above, device-level I/O is performed using a structure called IOREquest. The trackdisk.device uses an extended form of an IOREquest called an IOREgtd structure. Both of these structures are defined in the standard include file execio.h. In addition, the extended trackdisk-specific commands use an extended form of the IOStdReq structure. This extended IOREquest structure, called an IOREgtd structure and defined in the devices/trackdsk.h header file, is not used in the enclosed program, as the extended commands are not needed nor used in the program. Note that the ensuing discussion will refer to the IOREquest structure, without differentiating between these various extended versions.

**Copydisk**, the program

*Copydisk.c* was written to illustrate some of the principles involved with reading and writing floppy disks at the device driver level. It is a simple disk copier program, with a couple of advantages over the standard DiskCopy program that is distributed with the Amiga Workbench.

The program is designed to take advantage of the available memory in the system to allow you to make copies with a single drive with as few disk swaps as possible. Also, if your system has enough memory to hold the entire source disk's contents, you can make multiple copies of the same source disk, without having to re-read the source every time.

The program is designed to be run from the CLI only. The command line to invoke it is of the form:

```
 copydisk source destination [MULTI] [NOVERIFY]
```

where both the source- and destination-devices are valid disk drive identifiers (such as DF0: or DF2:).

The MULTI option, if specified, requests the program not to terminate after completing the first copy, but to continue prompting the user to allow multiple copies of the source disk to be made. This mode is terminated when the user enters a CTRL-C from the keyboard, as instructed in the prompts.

The NOVERIFY option will cause the data to be written to the destination disk much quicker, but provides a lot less error checking. It should only be used with caution, as it is quite possible for a write error to go undetected in this mode.

I decided to use some functions in the *Arp.library*, the freely-distributable library of support functions created and released by Charlie Heath, Scott Ballantyne and other. The program uses the Arp startup-code, as distributed with Arp Version 1.1 (the latest release available at the time I am writing this), instead of the standard startup-code distributed with the C compiler package. The advantage that this provides is that the Arp.library is automatically opened by the startup-code package, and the command line is parsed automatically, through a call to the Arp Gads() function, prior to control being passed to the main() function.

Consequently, the argument processing in the main() function can safely assume that the argument pointers in the argv[ ] array are in the order corresponding with the command-line-template, as specified in the CLI_Template variable.
The primary advantage in using Arp is the reduction in the amount of code that I had to write. The Arp startup allows me to have sophisticated command-line parsing and prompting, without having to add the code myself.

I also made use of the Arp memory-tracking (the ArpAllocMem() function), which allowed me to allocate the linked list of cache buffers, and bypass freeing them, since they are freed automatically during the program termination (thanks again to the closing of Arp library automatically by the _exit code contained in the Arp startup. Further savings was also realised by using the Arp function to perform the DOS packet-level IO required to stop the file-system from using a disk drive (refer to the inhibit_drives() in the accompanying program listing).

However, I did run into one potential problem using the Arp startup code. In the valid_devs() function, I attempt to walk the linked-list of device nodes in the system, in order to validate the devices specified in the command line. The way to find the beginning of this device list is through the RootNode structure, that is pointed to (via a BPTR) from the DsoLibrary. This is the standard DosBase pointer which is obtained by opening dos.library in the startup-code.

However, when using the Arp startup, DosBase really contains a pointer to the Arp library base (which contains a semi-clone of the DOS library). Unfortunately, this ‘clone’ is not an exact copy, and some fields contained in the DosLibrary structure are not set in the copy. In the 1.1 version of Arp library that I used, the BPTR to the RootNode was not set although I have been informed that with the 1.3 version of the library, which was in Beta testing when I wrote this, the RootNode pointer is available from this pseudo-DosBase pointer.

Note that the real DosBase pointer is saved in the ArpLibrary structure, and I needed to obtain that pointer (from the ArpBase pointer) in order to get the address of the RootNode. From there, I was able to find the system device node list, and perform the validations needed.

Examining the Disk I/O Routines

The first thing needed to access the floppy disk is to issue an OpenDevice() call for the device. This will initialise an IOResult structure with the values needed in order to perform I/O requests. The values set in this IOResult structure will be needed to send I/O requests to the driver. They could be copied to another allocated IOResult structure if needed (by copying the entire structure that is initialised by OpenDevice()). In our case, we will just initialise a single IOResult for the input device and use it for all of the input. The same process is followed for the output device.

We now need to determine the total number of tracks supported by the source device. This will tell us the maximum number of cache buffers we will allocate, since we wish to attempt to cache as much of the source disk as we can fit into memory (to minimise the number of disk swaps needed, if copying from and to the same drive, and to enable us to perform multiple writes of the disk from the single read copy of the source if possible).

We also will need to verify that the output device utilises the same number of tracks, as our logic depends on it. The TD_GETNUMTRACKS command in trackdisk will return the total number of tracks available on the disk unit in question. This value is returned in the io_Actual field in the IOREquest structure.

After we have successfully gained access to the device driver, we need to tell the AmigaDOS FileSystem to stay away from our disks. This is because it would be dangerous for the FileSystem to have access to the device while we have partially written to it, since we will not be maintaining the logical integrity of the file data between the time that we start writing to the destination disk and when we complete the copy operation.

This ‘keep your hands off’ request is handled by the routine inhibit_drives(), which utilises DOS Packet level IO to communicate with the FileSystem. Since this article is ostensibly about accessing the device driver rather than accessing the FileSystem handler, I'm not going to go into extensive detail about this. Suffice it to say that, by sending an ACTION_INHIBIT packet to the handler, with an argument value of TRUE, the FileSystem will consider the disk to be the standard 'Not a DOS Disk', and will not try to access it. The disk icon, if Workbench is loaded, will show DFH:BUSY.

After allocating the buffers we will need to cache the data read, we still need to allocate at least one buffer into which we will read the data and from which we will write the data. This is because the data area passed to the trackdisk.device for input and output operations must be in CHIP RAM. This is a requirement of the trackdisk, primarily because the device driver uses the Amiga's butter to perform the encoding and decoding of the data (the data that is physically on the floppy disk media is not stored as 8-bits of data per character as it is in memory, but is encoded using a technique called MFM, which helps to ensure that the series of hits is interpreted correctly). We aren't allocating our cache buffers specifically in CHIP memory, because it isn't possible to fit an entire floppy disk into the available CHIP area (until the very-fat-one is available).

This leads us to another design decision that I had to make in writing this program. I decided to use synchronous I/O, which means that my I/O requests will not return to my program until they have completed. This is accomplished by using the DoIO() function call. I could have chosen to perform asynchronous I/O, using the SendIO() function, in which case my program would get control after the request was accepted, but before it completed. That would enable me to, for example be reading one track while I was writing another (assuming that I was reading from and writing to different disk drives). I could also issue a second read request while the first one was processing.
but this latter would not result in much faster operation. While the first example (simultaneous reading and writing) would result in greater throughput, I chose not to implement it, primarily because it would add more complexity to the program, and I wanted to attempt to keep the example code simple and straightforward. I also needed to try to keep the size down, so the publishers wouldn't hang me.

As noted above, trackdisk supports reads and writes in any multiple of the sector size. However, since the data is transferred to and from the device in full tracks only, it is more efficient for us to design our input and output accordingly and read and write full tracks at a time (assuming we are always positioning to the offset that’s at the beginning of the track).

The reading of the data is pretty simple - we set up the request using the standard I/O command of CMD_READ, specify the beginning offset (expressed as a number of bytes) as the current track number (relative to zero) times the track size, and request to read a full-track length. This is read into our CHIP memory buffer and copied from there to our cache buffer.

Writing the track of data could have been done in an identical manner, using the standard I/O command, CMD_WRITE. The observant reader will notice, however, that the accompanying program does not use this technique. The copydisk program uses the trackdisk-specific command of TD_FORMAT instead.

Perhaps a bit of explanation is necessary. Because I could do a CMD_WRITE for a single sector of data, the way that trackdisk implements this is a little different to what one might expect. In order to write any data to a track on the disk, trackdisk first reads in the entire track from the disk, and then replaces within its internal buffer the sector(s) that the program is actually writing. It then will write out the full track from its buffer. In our case, it would read in the entire track, replace the entire track in its buffer with the data we are writing (since we’re writing the full track), and then write the data out. This works, as long as the disk we are writing to is already formatted - otherwise the initial read will fail and the write won’t be performed. However, it is considerably slower, since it requires the extra read of the full track before it is written.

There is an alternative, however, since we are performing full-track writes only. The TD_FORMAT command will simply write out a track’s worth of data, overlaying whatever happened to be on that track on the disk previously. This will work even for an unformatted disk, since this is the command that allows the disk to get formatted. The only restriction is that the operation must be performed in full-track increments.

The negative side to doing the writes with the TD_FORMAT command is that the command will appear to succeed, even if there is a problem with the disk (such as a bad disk). This will only be detected with a subsequent read of the track. This is why the program issues a CMD_READ request after the TD_FORMAT; unless the user decides to gamble by specifying the NOVERIFY command-line option. Note that with the verify, the operation is not any faster than the CMD_WRITE, except that we get a better test of the data on the output medium (since we read after we write). This technique will also work on an unformatted disk, whereas the CMD_WRITE technique would not.

One other command deserves some mention here: the TD_MOTOR command. As previously noted, this command is used to turn the floppy drive’s motor on or off. The observant reader will notice that the program listing never contains any I/O requests to turn the drive motor on. This is because trackdisk device will automatically turn the drive’s motor on when needed. However, it does not turn the motor off, unless explicitly requested by the application program. Consequently, we did have to include requests to turn the motor off at appropriate times, such as when we wish the user to replace the disk in the drive with another. It’s not a particularly good idea to leave the motor on (and the accompanying LED lit), and ask the user to swap disks. Note that we don’t automatically turn the motor off after each read or write, though, because that would slow things down. It does take a significant amount of time for the motor to get up to speed, and we only want to incur this overhead when it is really necessary.

Finally, it should be noted that, by doing a track-by-track copy of a disk, the resulting disk is an exact duplicate of the original. The Amiga File System keeps track of the disks it sees, and gets very confused if it sees two identical disks. It will generally cause your machine to either Guru or to lock-up. The way to avoid this is to ensure that the File System sees some difference in the two disks. To do this, we ensure that the disk creation date (contained in the root block of the disk) is different for the two disks. This can be seen in the code contained in the functions write_track() and touch_root_bkt() in the example program.

Source code with a dual personality

Copydisk was developed using Lattice C, version 5.02, in the standard 32-bit integer mode. However, it was intentionally written to be able to be compiled as-is using the current version (3.6a) of the Manx compiler, in its standard 16-bit mode.

Because of this design decision, certain coding constructs may appear to be a bit more complicated than those to which you are accustomed. For example, I definitely wanted to make use of the function-prototyping capability of the Lattice compiler, which is currently unsupported by AZTEC C. Consequently, I use the __ARGS() macro in declaring the prototypes. This macro is defined, for the Lattice package, in the stdio.h header file, as:

```c
#define __ARGS(a) a
```

and is defined (by me) for the Manx compiler (within an #ifdef AZTEC_C block) as:

```c
#define __ARGS(a) ()
```

This allows the function prototype statement in the form:

```c
int read_track __ARGS((struct cache *));
```

to be expanded, for Lattice, as:

```c
int read_track (struct cache *);
```

and, for Manx, as:

```c
int read_track ();
```

which are both forms that the respective compilers will accept as a forward reference/definition of a function.

The difference in integer size in the default modes of both compilers caused a few incompatibilities. This resulted in some extra casts that might appear unnecessary in one implementation or the other, but allow the source code to work correctly with both systems.

In Conclusion

`Copydisk.c` illustrates some techniques used in accessing the floppy disk drives at the device driver level. It also is a useful disk-copier program, although that is a secondary benefit to its reason for existence. Because of this, there are some shortcomings to the code as presented. There are also some opportunities (left to the reader as a homework exercise) for substantially improving the program and its capabilities.

Better error-handling logic is needed while desirable enhancements include the adding of asynchronous IO logic (which is not quite as trivial as it may at first seem).

Also on my wish-list is a better interface, including a real Intuition-style interface so that the program is not restricted to CLI use, but it's a start.
void main(argc, argv)
{
  src = argv;
  dst = argv;

  if (argc - 1) {
    printf("Usage: \%s argv[0], CLI_Template\n", argv[0], argv[1]);
    printf("CLI_Help\n", argv[0]);
    exit(1);
  } /* Save the pointers to the source/dest drive names */

  src_name = argv[ARG_FROM];
  dst_name = argv[ARG_TO];

  if (argc[ARG_MULTI])
    do_multi = TRUE;
  if (argc[ARG_VERIFY])
    do_verify = TRUE;
  if (argc[ARG_VERIFY])
    do_verify = TRUE;

  if (!init_devs()) /* If error during initialization */
    cleanup();
  exit(0);
}

void cleanup()
{
  ret = do_copy();
  cleanup();
  exit(0);
}
function: do_copy()

Purpose: controls the high-level reading and writing of disks

Inputs: none

Outputs: The (eventual) program return code

Notes:

Only checks for ctrl-C interrupts when the user is being prompted.

int do_copy()
{
  struct cache *buf_ptr;
  while (TRUE)
  {
    if (Have_src_disk)
    {
      /* prompt for source disk to be inserted */
      PRINTV(srcprompt, src_name);
      if (disk_unit != idsk_unit) /* if not single-drive copy */
      {
        PRINTV(dstprompt, dst_name);
      }
      PRINTV(prompt); Readline(stdin);
      if (CheckAbort(NULL)) /* check for ctrl-C */
      {
        return(0);
      }
      have_src_disk = TRUE;
      cur_trk = par_trk = 0;
      while (cur_trk < max_trk)
      {
        if (!disk_in_mem) /* if we haven't buffered the src disk yet */
        {
          buf_ptr = trk_buf; /* point to first cache */
          while (buf_ptr < max_trk)
          {
            if (read_trk(buf_ptr))
            {
              if (write_trk(buf_ptr))
              {
                PRINTV("Unable to write destination disk\n");
                return(10);
              }
            }
            if (buffers == max_trk) /* next in linked-list */
            {
              buf_ptr = buf_ptr->next;
            }
            if (inum_trks == max_trk) /* we've cached the entire disk */
            {
              stop_drives(); /* turn the motors off */
              if (disk_unit == idsk_unit)
              {
                /* single-drive copy - can't have source * and dest both present */
                PRINTV(srcprompt, src_name);
                PRINTV(dstprompt, dst_name);
                PRINTV(prompt);
                Readline(stdin);
                if (CheckAbort(NULL))
                {
                  return(0);
                }
                have_src_disk = FALSE;
              }
            }
            else
            {
              /* only checks for ctrl-C interrupts when the user is being prompted */
              if (disk_unit == idsk_unit)
              {
                /* single-drive copy */
                PRINTV("Disk copied\n");
                if (do_mult)
                {
                  break; /* exit the while(TRUE) if simple copy */
                }
                PRINTV(srcprompt, src_name);
                PRINTV(dstprompt, dst_name);
                Readline(stdin);
                if (CheckAbort(NULL))
                {
                  break;
                }
              }
            }
          }
        }
        else /* if we haven't written the */
        {
          /* entire disk */
          if (disk_unit == idsk_unit)
          {
            /* single-drive copy */
            PRINTV("Single drive copy\n");
            if (do_mult)
            {
              break; /* exit the while(TRUE) if simple copy */
            }
            PRINTV(srcprompt, src_name);
            PRINTV(dstprompt, dst_name);
            Readline(stdin);
            if (CheckAbort(NULL))
            {
              break;
            }
          }
        }
      }
    }
  }
}

function: init_dev()
int init_disk() {
    long totmem;
    int i;

    struct cache *cache_ptr;

    /* Note that if the drives have been specified using */
    /* a trailing colon, as is the standard name of */
    /* specifying a device name, this colon is stripped */
    /* off. This is because the device name will be */
    /* compounded with the device name as contained in the */
    /* AmigaOS system list of devices, and the name is */
    /* that list do NOT contain the colon. */
    /\n    i = strlen(src_name);
    if (src_name[i-1] == ':') {
        src_name[i-1] = '\0';
    }
    i = strlen(dest_name);
    if (dest_name[i-1] == ':') {
        dest_name[i-1] = '\0';
    }

    if (!valid_dev() /* insure that the devices specified */
        && !valid_devname, and any */
        return(FALSE); /* actual "trackdisk" device" */

    if (!open_tdisk()) /* open the devices */
        return(FALSE);

    /* Keep the file system from accessing the drives */
    if (!inhibit_drives(1))
        return(FALSE);

    /* NOTE that trackdisk requires its buffer to be in */
    /* CLIP memory */
    trk_buf = (char *)ApptlbMem(PULL TRASH, /*
     * NORMAL CLR */ NORMAL_PUBLIC | NORMAL_CLEAR);

    if (trk_buf) {
        Print("Not enough CHIP MEMORY to Proceed!");
        return(FALSE);
    }

    /\n    /* We want to allocate enough cache buffers to hold the entire */
    /* source diskette in memory, if possible. We will, however, */
    /* insure that at least 64K of memory is left unallocated. */
    /* This will insure that the system has enough memory to handle */
    /* most transient requirements. */
    /*
    totmem = (long)AvailMem();
    totmem -= 512K; /* leave at least 64K available to the system */
    if (totmem > (sizeof(struct cache) << 1)) /* enough far at least two */
        totmem /= sizeof(struct cache);
    if (totmem > max_trk) totmem = max_trk;
    for (num_buf = 0; num_buf < totmem; num_buf++) {
        cache_ptr = (struct cache *)
            AppTmpMem((long)sizeof(struct cache), (long)NORMAL_CLEAR);
        if (!cache_ptr)
            break;
        cache_ptr->next = trk_buf;
    }
}

/* Function: valid_dev() */
/* Purpose: Verifies that the devices requested for source and */
/* destination are valid devices in the system, and */
/* are, in fact, controlled by trackdisk device */
/* Inputs: Global: source-name, destination-name */
/* Outputs: Boolean success indicator */
/* Global: device/destination unit numbers */
/* Notes: */
/* This routine walks through the system-level linked list */
/* of device nodes. It must do this with task-switching */
/* disabled, to insure that this chain doesn't change while */
/* we are inspecting it. */
/* */
/* NOTE that the DosLibrary structure, which is used to */
/* locate the device node list, is normally pointed to by the */
/* standard DosBase pointer. However, because we are using */
/* the ARP startup, DosBase really points to Arpsave, and not */
/* all of the DosLibrary structure is valid. We have to use */
/* the REAL DosBase pointer, which is conveniently saved for */
/* us in the Arp Library structure. */
*/

int valid_dev() {
    char src_buf[23],
        dev_buf[64];
    int src = FALSE;
    struct DosLibrary *real_dos;
    struct DeviceNode *dlist;
    struct RootNode *rn;
    struct FileInfo *dl;
    struct FileSysStartupInfo *fs;

    trk_buf = (char *)ApplMem(PULL TRASH, /*
     * NORMAL CLR */ NORMAL_PUBLIC | NORMAL_CLEAR);

    if (trk_buf) {
        Print("Not enough CHIP MEMORY to Proceed!");
        return(FALSE);
    }

    /* We want to allocate enough cache buffers to hold the entire */
    /* source diskette in memory, if possible. We will, however, */
    /* insure that at least 64K of memory is left unallocated. */
    /* This will insure that the system has enough memory to handle */
    /* most transient requirements. */
    /*
    totmem = (long)AvailMem();
    totmem -= 512K; /* leave at least 64K available to the system */
    if (totmem > (sizeof(struct cache) << 1)) /* enough far at least two */
        totmem /= sizeof(struct cache);
    if (totmem > max_trk) totmem = max_trk;
    for (num_buf = 0; num_buf < totmem; num_buf++) {
        cache_ptr = (struct cache *)
            AppTmpMem((long)sizeof(struct cache), (long)NORMAL_CLEAR);
        if (!cache_ptr)
            break;
        cache_ptr->next = trk_buf;
    }
}

/* Function: valid_dev() */
/* Purpose: Verifies that the devices requested for source and */
/* destination are valid devices in the system, and */
/* are, in fact, controlled by trackdisk device */
/* Inputs: Global: source-name, destination-name */
/* Outputs: Boolean success indicator */
/* Global: device/destination unit numbers */
/* Notes: */
/* This routine walks through the system-level linked list */
/* of device nodes. It must do this with task-switching */
/* disabled, to insure that this chain doesn't change while */
/* we are inspecting it. */
/* */
/* NOTE that the DosLibrary structure, which is used to */
/* locate the device node list, is normally pointed to by the */
/* standard DosBase pointer. However, because we are using */
/* the ARP startup, DosBase really points to Arpsave, and not */
/* all of the DosLibrary structure is valid. We have to use */
/* the REAL DosBase pointer, which is conveniently saved for */
/* us in the Arp Library structure. */
*/
if (Strcmp(src_name, nm_buf) == 0)
    { dsk_unit = fs->fss_Unit; /* grab the unit */
    }
if (Strcmp(dst_name, nm_buf) == 0)
    { odsk_unit = fs->fss_Unit;
    }
else /* the device is not trackdisk.device */
    { Permit(); /* remember to enable multitasking */
      printf("Error: Source device not floppy drive\n", "Source": "Destination");
      return (FALSE);
    }

dlist = (struct DeviceNode *) BDOR(dlist->dlist.Next); /* next in chain */
Permit(); /* multitasking ON again */

/* NOTE: the device unit numbers are initialized to -1 */
/* if that value hasn't changed, then no matching entry */
/* was found for that device name in the list. */
if (!dsk_unit || dsk_unit == -1L)
    { printf("Error: Source device not found\n");
      err = TRUE;
    }
if (!odsk_unit || odsk_unit == -1L)
    { printf("Error: Destination device not found\n");
      err = TRUE;
    }
if (err) return (FALSE);

printf("Not Enough Memory to Proceed\n");
return (FALSE);
int open_td()

if (!td_port)
    {
      printf("Unable to allocate a Port/Signal Eit\n");
      return (FALSE);
    }

if (OpenDevice (TD_NAME, dsk_unit, (struct IORequest *)tdireq, 0L))
    { Permit(); /* remember to enable multitasking */
      printf("Can't allocate Source Device\n");
      return (FALSE);
    }

if (OpenDevice (TD_NAME, odsk_unit, (struct IORequest *)tdireq, 0L))
    { Permit(); /* remember to enable multitasking */
      printf("Can't allocate Target Device\n");
      return (FALSE);
    }

td_open = TRUE;

/* Function: inhibit_drives() */
/* Purpose: Either dis-allow the file-system from using the */
/* drives, or re-allow standard file-system access */
/* Inputs: Global: device names */
/* Outputs: Boolean success indicator */
/* Notes: */
/* Uses DOS packet-level IO. */
/* When the devices are "inhibited", they are treated as */
/* "Not a DOS disk", and will show as OPS:BUSY on the NB */
/* screen. */

int inhibit_drives(int i_flag)

char dev_name[40];
int i;
long pkt_args[4];
struct MapPort *td_map Proc,

sprinf(dev_name, "%s", src_name); /* insure the name ends with */
/* * Get the address of the process's MapPort structure for */
/* the device driver, */
/* tmap_proc = (struct MapPort *)DeviceProc(dev_map); */

/* Purpose: Open the trackdisk.device units, allocate the IO */
/* request structures, get/validate disk information */
/* (total number of tracks). */
/* Inputs: Global: unit numbers (src and dst) */
/* Outputs: Boolean success indicator */
/* Global: input and output IO request structures */
/* Notes: */
/* We only will perform the copy if the source and destination */
/* units have the same maximum number of tracks. */
/* won't, for example, copy from a 5-1/4" floppy to a 3-1/2" */
/* drive. */

int open_td()

{ td_port = (struct MapPort *) CreatePort (0, 0);
if (!td_port)
    { printf("Unable to allocate a Port/Signal Eit\n");
      return (FALSE);
    }
if (idev_proc) {
  printf("Can't locate Source Device's processes\n");
  return(FALSE);
}
/*
 * If the input and output devices are the same, the processes
 * are the same. Otherwise, repeat for the output device
 */
if (idisk_unit == odisk_unit) {
  idev_proc = idev_proc;
}
else {
  printf("Can't locate Target Device's processes\n");
  return(FALSE);
}
for (i=0;i<2;i++) {
  pkt_args[i] = 0; /* initialize the packet argument array */
  pkt_args[i] = i_flag; /* First argument is a Boolean value */
  if (idev_owned || odisk_owned) {
    idev Owned = TRUE;
  }
  if (idisk_unit == odisk_unit) /* only if two devices */
    pkt_args[0] = i_flag;
  if (idev_proc) {
    printf("Unable to inhibit source drive\n");
    return(FALSE);
  }
  if (i_flag) {
    idev Owned = TRUE;
  }
  return(TRUE);
}

printf("Reading Cylinder %ld Read %ld \n", 
  (long)(idisk_trk >> 1), 
  (long)(idisk_trk & 1));
for (retry=0;retry<3;retry++) /* up to 3 retries */
  td ireq->io Command = CMD_READ;
  td ireq->io Length = FULLTRACK;
  td ireq->io Offset = (long)(idisk_trk * FULLTRACK);
  /*
   * MTR that trackdisk reads west be to a buffer
   * that is in CRIB memory. Consequently, we can't
   * read directly into our cache buffer.
   */
  td ireq->io Data = (AMIGA) td buf;
  i = ioff(irequests * ireq)->io Offset;
  if (i) /* success */
    if (idisk_tkrk == 0)
      if (!SendPacket(ACTION_INHIBIT,pkt_args,idev_proc)
        printf("Unable to inhibit source drive\n");
        return(FALSE);
      else
        idev Owned = TRUE;
    else
      idev Owned = TRUE;
  return(FALSE);
}

int read_track(dbuf)
  struct cache *cbuf;
  long i;
  int entry;

  printf("Reading Cylinder %ld Read %ld \n", 
    (long)(idisk_trk >> 1), 
    (long)(idisk_trk & 1));
  for (retry=0;retry<3;retry++) /* up to 3 retries */
    td ireq->io Command = CMD_READ;
    td ireq->io Length = FULLTRACK;
    td ireq->io Offset = (long)(idisk_trk * FULLTRACK);
    /*
     * MTR that trackdisk reads west be to a buffer
     * that is in CRIB memory. Consequently, we can't
     * read directly into our cache buffer.
     */
    td ireq->io Data = (AMIGA) td buf;
    i = ioff(irequests * ireq)->io Offset;
    if (i) /* success */
      if (idisk_tkrk == 0)
        if (!SendPacket(ACTION_INHIBIT,pkt_args,idev_proc)
          printf("Unable to inhibit source drive\n");
          return(FALSE);
        else
          if (!SendPacket(ACTION_INHIBIT,pkt_args,idev_proc)
            printf("Unable to inhibit source drive\n");
            return(FALSE);
NOTES: AmigaOS really doesn't like two DOS disks in the same system that are absolutely identical. This will cause the system to lock up or GUM if both disks are present at the same time, if the file-system has access to the disks.

We will 'tweak' the root block of the output disk, to avoid the 'identical disk' syndrome.

Purpose: 'Tweaks' the root block, adjusting the 'data created' field, so that this disk won't be identical to the disk we read.

Inputs: Pointer to the root block image

Outputs: none (Root block image is updated)

We verify that the block passed is indeed the root block.

The AmigaOS Tech. Ref. Manual documents the format of the root block in terms of long-words. The values used in this routine match that documentation.

The checksum for the block has to be re-calculated. The checksum is the value that causes the sum of all the long-words in the sector (including the checksum) to total a value of 0.

Function: touch_root_blk()

Purpose: 'Tweaks' the root block, adjusting the 'data created' field, so that this disk won't be identical to the disk we read.

Inputs: Pointer to the root block image

Outputs: none (Root block image is updated)

Notes:
The trackdisk.device will always turn a drive's motor on when it is needed (for a read or write), but does NOT turn the motor off. There is no harm done in attempting to turn off the motor if it is already off, so we don't really need to keep track of the motor's state, just turn it off when we want to insure it's off.
void stop_drives() {
    if (tdin_open & (tdout_open))
        stop_drives();
    if (tdin_open)
        CloseDevice(struct IDRequest *td_inreq);
    tdin_open = FALSE;
} // if (tdin_open)
CloseDevice(struct IDRequest *td_inreq);
    tdin_open = FALSE;
} // if (tdin_open)
if (idev_owned || odev_owned)
    inhibits_drives();
    idev_owned = odev_owned = FALSE;
} // if (idev_owned || odev_owned)
if (td_inreq)
    DeleteStdC(td_inreq);
    td_inreq = NULL;
} // if (td_inreq)
if (td_outreq)
    DeleteStdD(td_outreq);
    td_outreq = NULL;
} // if (td_outreq)
if (td_port)
    DeletePort(td_port);
    td_port = NULL;
} // if (td_port)

void cleanup() {
    if (tdin_open && tdout_open)
        stop_drives();
    if (tdin_open)
        CloseDevice(struct IDRequest *td_inreq);
    tdin_open = FALSE;
} // if (tdin_open)
CloseDevice(struct IDRequest *td_inreq);
    tdin_open = FALSE;
} // if (tdin_open)
if (idev_owned || odev_owned)
    inhibit_drives();
    idev_owned = odev_owned = FALSE;
} // if (idev_owned || odev_owned)
if (td_inreq)
    DeleteStdC(td_inreq);
    td_inreq = NULL;
} // if (td_inreq)
if (td_outreq)
    DeleteStdD(td_outreq);
    td_outreq = NULL;
} // if (td_outreq)
if (td_port)
    DeletePort(td_port);
    td_port = NULL;
} // if (td_port)

Makefile for Lattice compiler

DBG = 1
LTP = SC M0

COPYDISK : copydisk.o
    blink from lib:arpc,etcopydisk.o to copydisk lib library, lib libclib.lib 
    libarpc.lib $(LTP)

copydisk.o : copydisk.c
    %CFLAGS -
    in -o copydisk.o -larpc(LTP) -lcl(LTP)

Makefile for Manx compiler

# Manx 43.6 makefile for copydisk.

CFLAGS -

COPYDISK : copydisk.o
    %CFLAGS -
    in -o copydisk.o -larpc(LTP) -lcl(LTP)
Announcing . . . WShell
THE COMMAND SHELL YOU'VE ALWAYS WANTED

The WShell is a powerful command shell that gives you the features you always wanted—but didn’t get—in the CLI. It has command aliases, built-in commands, resident commands, a configurable titlebar, and concurrent command piping. And yet it’s highly compatible with the CLI, so you don’t have to learn a new command language to use it.

WShell was written by the author of ConMan, one of the Amiga's most popular shareware products. It includes the latest version of ConMan—with more editing features and fast forwards/backwards searching.

- Command Aliases and Abbreviations
- Convenient Set of Built-in Commands
- User-Extensible Resident Commands
- Configurable Prompt String/Window Titlebar
- Command Piping Using the Familiar "I" Syntax
- ARexx Interface for REXX-Language Macros
- Script-Bit Support for Batch File Execution
- Compact, Shared Code—Less Than 12K

Command Piping

The WShell supports concurrent command piping, just like the well-known operating system with the trademarked name. A command like

```
program1 -x args I program2 -y I program3 -z
```

results in a system of three shells running three programs concurrently with each program feeding its output to the next program's input.

ARexx Interface

WShell becomes a fully programmable command shell when combined with ARexx, the REXX language for the Amiga. REXX is a high-level structured language that supports recursive function calls and even includes a full source-level debugger. WShell transparently executes REXX-language macro programs, so you can use macro programs just like an executable—even as a filter program in a piping system. Use the source-level debugging to single-step through a complex system of concurrently-running "piped" programs!

THE PRICE? . . . ONLY U.S.$50!

Whether you're a weekend user or a professional programmer, WShell will help you get the most from your Amiga! Look for the WShell at better dealers, or order by mail from the address below. On mail orders please include $2 for shipping ($8 for airmail outside of U.S.) plus any applicable taxes.

DEVELOPED AND SUPPORTED BY:

William S. Hawes
PO Box 308
Maynard, MA 01754
(508) 568-8695

REQUIRES AN AMIGA 500/1000/2000 WITH V1.2 OS • AMIGA IS A TRADEMARK OF COMMODORE-AMIGA, INC.
An interactive disassembler for the Amiga computer. Written entirely in hand-coded assembly language, this is the most powerful disassembler available on ANY personal computer!

Over 700 menu functions allow you to:
> Load from ANY file, read tracks, or disassemble memory directly. > Save ALL, allowing you to continue disassembling later. > Save to a "$asm" file, partial saves supported. > Automated symbol creation IE. "JSR -$1E(A6)" becomes "JSR _LVOOpen(A6)". "MOVE.L #$3EE,D0" becomes "MOVE.L #MODE_NEWFILE,D0". Over 5000 symbols supported! > Incredibly smooth scrolling. > Full key rebinding supported. Load/save keytables. > Up to 38 user-defined macros available. > Numeric and text processing functions allow you to speed up disassembling. > Search function uses full ARP wildcards. Search forwards, backwards, or bi-directional. > The fastest text rendering available on the Amiga. > Forwards and backwards referencing allows you to quickly examine referenced code/data, and return to your original position. Nested references supported. > Automatically/manually create labels. Manually edit/delete labels, symbols, end-of-line comments, full-line comments. > Special support for "C" programs, including "A4" register data referencing.

If you’re serious about disassembling, look no further!
Available by mail order from the author (A$95)

Glen McDiarmid
28 Marginson St
Ipswich QLD 4305
(07) 812-2963

In the USA, contact:
The Puzzle Factory
P.O. Box 986
Veneta OR 97487
(503) 935-3709

(Credit card facilities available)