- Faster, and more efficient assembly code by John Toebes
- Build a MIDI interface - circuit diagram & software by Steve Simpson
- A faster Flood-fill by Danny Ross
- Algorithmically generated Gadgets by Peter Booth
- Assembly Language Programming - part 2 by Jim Butterfield
- BreakPoint - fifth in a series on debugging by Victor A. Wagner
- Driving the Drives by Dr. Rahman Haleem of Commodore (UK)
- Hard disk backup utilities reviewed by Steve Ahlstrom
- Programming in ARexx - part 2 by John Carpenter
- Plus regular columns by Larry Phillips, Don Curtis and Steve Ahlstrom

Woodland2 - by Louis Markoya
SPIRIT TECHNOLOGY
Superior Expansion Hardware for COMMODORE AMIGA

**IN1000: 1.5 MB for AMIGA 1000**
- Utilizes 256K by 1 DRAM.
- Expandable from 0K to 0.5, 1.0 and 1.5 MB.
- Includes battery backed Clock/Calendar.
- Full memory auto-config with RAM on/off.
- Adds up to a full 1.5 MB memory to existing Amiga RAM.

**PRICE** $395.00 OK

**IN500: 1.5 MB for AMIGA 500**
- Utilizes 256K by 1 DRAM.
- Expandable from 0K to 0.5, 1.0 and 1.5 MB.
- Operational auto-config with RAM on/off.
- Ad-mem adds a full 1.5 MB to existing RAM.
- Optional battery backed Clock, Calendar chip.

**PRICE** $395.00 OK

**SIN500: 2 MB for AMIGA 500**
- Utilizes 1.0 Mbit (256 by 4) DRAM.
- Expandable from 0K to 0.5, 1.0 and 2 MB.
- Full memory auto-config with RAM on/off.
- Adds up to a full 2 MB memory to existing Amiga RAM.
- Optional external power supply.

**PRICE** $395.00 OK

**SC501: AMIGA 501 CLONE**
- 0.5 MB Expansion for Amiga 500.
- Built in Clock/Calendar.

**PRICE** $109.00 OK
$239.00 WITH 0.5 MB

**ADD ON RAM, SOLD ONLY WITH SPIRIT MEMORY BOARDS**
$150.00 FOR 0.5 MB

**HDA-506: IBM Hard Drive Adaptor**
- Adapts all ST-506 hard drives, MFM or RLL.
- Amiga 86-pin expansion port passthru.
- Includes Custom Driver Software, compatible with FFS 1.3 and low level formatter.
- Optional 1.3 Autoboot EPROM.
- For the Amiga 1000/500.

**PRICE** $395.00

**MS-2X6: MIDI Console**
- 2 MIDI IN's with switch control and LED status.
- Amiga RS-232 serial interface with passthru.
- 6 MIDI OUT/OFF/THRU's, switch control and multi-colour LED status.
- 2 separate operational modes.
- Compatible to all RS-232 C MIDI computers.

**PRICE** $395.00

12 MONTHS WARRANTY ON ALL SPIRIT EXPANSION PRODUCTS

NEW . . . SPIRIT ST-506, 3.5" HARD CARD FOR AMIGA 2000!
$429.00

ORDER NOW.
PHONE OR
MAIL. WARD
OVERNIGHT
DELIVERY

POWER PERIPHERALS
PO BOX 555 LAVERTON MELBOURNE VICTORIA 3028 PHONE (03) 369 7020 FAX (03) 369 7020
G'DAY

Transactor for the Amiga has just had its first birthday, and its nice to be able to say that we are doing fine. To you, our readers, and the dealers and advertisers who support us, thank you.

For myself, my connection with Commodore now covers ten years. Its interesting to muse on how the computer world generally, and Commodore specifically, has changed. We have come from infancy in the late seventies to maturity in the late eighties, with a totally different concept of design, marketing and support. Among users, of course, there has been a quantum leap in computer literacy and expertise. It would be a foolish manufacturer who ignored that.

What about the nineties? My vision of what lies ahead is probably no clearer than yours, but the Amiga has now been around for a while, which begs the question of... what next? Someone in Commodore must be thinking ahead, trying to figure out the specifications for the products to sustain the company into the next decade. What a difficult task—and what a difficult decision to contemplate! But its probably sometime about now that this sort of planning has to happen.

For the Amiga, there is likely to be a strong move towards providing the technology to permit/encourage integration of Amiga with other Amigas, and also into the MS-DOS/Apple/??? world. To some extent, Commodore are the last on the block to do this. Assuming that such is their desire, and they do make it possible, will the market-place respond?

So much for trying to look over the horizon. Back here on earth, a boring but necessary housekeeping chore, directed to those of you who subscribe directly to us. The mailing labels now show you the last issue of your current subscription—take a look and check now. When the fateful day comes around, you will find a renewal form inside your magazine, to remind you that the time has come. Just fill in the details on the form, and mail it back. Easier still, ring (7-9pm weekdays) and we can take care of you.

We have offered space to Commodore to keep you in touch with their doings. The offer still stands, so maybe next issue...

There has been a strong reaction to direct sales of software by overseas outlets. Here in Canberra, not one but two agencies have opened for business, and offer software at quite good prices. Details appear later in this issue.

And last of all, my quest for a word-processor for the Amiga has been decided. What did I choose? Read on into this issue, and all will be revealed.

Paul Blair
ViewPort
by Larry Phillips
Commodore provided Microsoft's AmigaBASIC with every Amiga purchased - but it really doesn't live up to expectations. HiSoft have produced a compiler for AmigaBASIC which makes it bearable. Larry takes a sideways look at this and the philosophy of Amiga programming languages.

Dispatches
by Don Curtis
Why don't you give your Amiga a spring-clean? It might be a little late in the year, but Don's had a go, and tells his tale! Not satisfied with cleaning his Amiga, he goes on to tell the tale of a broken Bridge-board and the hazards of getting it fixed.

Driving the Drives
by Dr. Rahman S. Haleem
A late submission (and a knock out .... oops, sorry!) from Commodore's Technical Support Manager on the dangers of driving without due care and attention.

Access - the best of non-commercial software
by Steve Ahlstrom
AmigaZoo is an archiving program, much like ARC, but faster - our explorer in the world of non-commercial software takes a closer look at its advantages and disadvantages. He also examines SetCPU, a utility for the rich who have fancy devices like 68020, 68030, 68851, 68881 on their A2620 boards. Assign extra paths to system directories, such as FONTS: or DEV$: with PathAss. Talk to the world with JrComm - and there's some news on the SoftCircuit CAD package discussed by Steve earlier in the year.

Assembly Language, Part 2
by Jim Butterfield
In part 2 of this series on programming the Amiga in assembly language, our guiding Guru looks at the command line arguments, and puts a twist in ECHO's tail.

BreakPoint, Part 5
by Victor A. Wagner
In this final installment aimed at the gentle art of debugging, Vic Wagner describes two more debugging tools: DB, from Mars, and MetaScope from Metadigm. Vic unashamedly declares an interest in the latter, for he has helped develop the program and wrote the demonstration text which comes with the package.

Cover Picture: Woodland2
by Louis Markoya
This front cover probably represents the most complex and sophisticated ray trace done on the Amiga to date. The project was started with the desire to create organic objects with lifelike quality, to produce photo realism in Amiga ray tracing beyond chrome spheres and checkered floors. The scene is 100% ray traced with Turbo Silver. The objects represent a months work to produce with Turbo Silvers editor. I feel this picture shows the Amiga can produce results usually reserved for much higher end systems, opening the door to further explore and stretch the possibilities of the Amiga.
Multiple Gadgets
by Peter Booth

Some applications demand many on-screen gadgets. Take an on-screen roulette wheel, for instance, and in order to place gadgets around its perimeter it is most useful to be able to create a single gadget, and from this, generate multiple gadgets positioned according to a formula. Peter puts forward a few ideas which might help in this, and many other applications.

MIDI - the hardware
by Steve Simpson

This is the first of several articles relating to programming and using the MIDI interface. Steve Simpson starts us off with a look at the hardware required and presents a circuit diagram for an Amiga MIDI interface.

The Art of Assembly Language
by John Toebes

John is one of the leading Amiga assembly language programmers, having authored the latest Latte C compiler. In this article, he offers an eighteen point plan for making your code faster, shorter, but above all, more efficient. Buzzwords such as 'peephole optimisation', 'local optimisation' and 'global optimisation' are the hallmark of sophisticated assembly language programs... and you'll find them all here.

Hard Disk Backup
by Steve Ahlstrom

Steve stays off the path of non-commercial software for a while to take a look at three commercial products which are designed to back-up hard drives: Quizmaster, from Central Coast Software; Safe-Net, from Metadigm Inc.; and ExpressCopy, from ExpressWay Software. They are all designed to give you peace of mind, but just how well does each perform? Read on....

ARexx Programming
by John Carpenter

In this, the second in his occasional series, John Carpenter describes decision making in ARexx with IF...THEN...ELSE statements, several looping variations on the DO command and a brief look at the ARG and INTERPRET commands.

A Fast Flood Algorithm
by Danny Ross

Fast filling of rectangles using the Amiga's blitter is second to none in performance. Using the AreaFill library routine, squares and oblongs can be zapped to the screen at an incredible rate. But, try to fill an irregular area with the Flood routine and you'll wait for ages. Danny Ross describes a much more efficient Flood routine using the Line Adjacency Algorithm, and an efficient piece of assembly language. There's also a demonstration program to show just how fast his routine really is!
In praise of Basic ... but not AmigaBasic

by Larry Phillips
Copyright © 1989 Larry Phillips

If you ask an Amiga owner about his computer background, the response could be nearly anything. From first-time computer owners to those with extensive experience in mainframes, from system-programmer to end user, from beginner to expert, they come from all backgrounds.

Many are served well by the environment they find on the Amiga, with applications or programming languages.

The programming side of the Amiga owner base has many tools to call upon, including assemblers or compilers for C, Fortran, Lisp, Pascal, Modula-2, APL, and so on. Application programs abound, covering many interests. There is, however, one area that could be improved upon.

Not everyone likes to program, and there are those who shudder at the very thought of writing any kind of program, even simple script or batch files. The Amiga owner in this position must either be content with existing programs, wait the development of new applications, or actively convince a programmer that the desired application is a desirable one.

At the other end of the programming spectrum are the true programmers, the developers of application software, utilities, games, and more. These are the 'language junkies' who try any language available to them. In fact, some of these people have computers for no other reason than to write programs. They are the people responsible for writing the many programs we find on the Amiga.

They are not about to write programs that they don't have a reason to write. It may be that they have a personal need for a particular program, or that they may gain some financial benefit from the program.

Between these two extremes, you will find many Amiga owners, with a full range of ability and desire to create programs. That is where things could be improved upon.

There are a very large number of Amiga owners that would, given the tools, write small, simple programs. They would write the little utilities that nobody else seems to want to write. These could be the one-shot programs, the very specialised programs that might only be useful to one person. They could be more ambitious, growing over time to become full-blown, generally useful programs. Not everyone wants to have to buy a compiler or learn a new language in order to do this.

Imagine the person coming from the C64, Atari 800, IBM PC, or any other small computer, having written a number of programs in BASIC, and wanting to do the same on the Amiga. They open the Amiga box, set everything up, and the first time they want to write that little utility, they come smack up against AmigaBASIC.

I don't know what the BASIC environment is like on the IBM PC or the Atari (or most other 8 bit micros), but I can tell you that BASIC environment on the C64 is pure luxury when compared with that of AmigaBASIC.

Now the owners who want to write the occasional program because they need it, might put up with the deficiencies in the user interface of AmigaBASIC, but would the person writing things for fun put up with it?

It sure isn't much fun watching the windows redraw, or realising right after you run a program, that you have a syntax error, and are faced with the painful job of waiting for the requester to appear, with its one and only one choice, reaching for the mouse, cancelling the requester, and waiting for the error window to finally go away.

AmigaBASIC in itself, as I have said before in the ViewPort, is a decent implementation of BASIC, but only if you can ignore the bugs and the pitiful user interface. The only reason I mention it again is because I have seen a BASIC that is easy to use, friendly, and is compatible with AmigaBASIC (minus all the bugs of course).

This BASIC is from HiSoft, a company which also produces software for Atari computers. The BASIC is called 'HiSoft BASIC Professional', and it lives up to its name in more ways than one.

HiSoft BASIC provides an environment that is pleasing to use, with an integrated editor and error finder, allowing you to edit,
compile, correct errors, and test the program without leaving the editor.

It also allows you to write code that needs a resident library, or code that is fully stand-alone. The stand-alone code is fairly small, with the run-time functions being extracted from the library only if needed, and library dependent code is very small, the run-time functions being provided by a 46K shared library.

You are not restricted to their editor and environment, so you can use the editor of your choice and the CLI to write programs.

It comes with a thick manual that is accurate and readable, and all in all, is a very well thought out and implemented package.

It's difficult to talk about HiSoft without doing so in glowing terms. Though I am not really a BASIC fan, I can appreciate a job well done in any given language implementation. It's also difficult to talk about it without giving it a mini-review, but I wanted to let you know the differences between AmigaBASIC and the HiSoft package before going on.

Given the enormous difference in useability between these two BASIC packages, it pains me to see AmigaBASIC still being provided with the machine. It pains me to see Microsoft profiting from a half-baked effort. It pains me to think of all the Amiga owners who, if it were not for AmigaBASIC, might be writing any number of programs, perhaps to the benefit of the Amiga community in general, and most certainly to their own benefit. (It is perhaps significant that there is very little in the way of 'freely redistributable' AmigaBASIC software available, and, as a language, it is shunned by many in the serious computing community - ED)

A year or so ago I asked Microsoft, through their on-line forum on Compuserve, if they were going to upgrade their Amiga product. The answer I got was essentially that I should ask Commodore that question. The distinct message, though not explicitly stated, was that they were simply not interested.

Now, in fairness, it might well be that they delivered on their contract with Commodore, that the product was accepted, and that in order to upgrade, Microsoft would require further payment.

I can't imagine any other reason for a company to leave a program like AmigaBASIC unchanged, with all that implies in company image. Perhaps they don't care that all Amiga owners will know just how bad a Microsoft product can be, or perhaps they don't think there will ever be enough Amiga owners to make a difference.

In thinking about it, I am not even sure that I would want Commodore to pursue the matter, given the alternative of packages such as HiSoft Professional, and given Microsoft's past efforts on other machines. I do think, however, that Commodore might do well to consider licensing a package like HiSoft Professional for inclusion with every machine. Consider the advantages that would be had in providing a truly good BASIC.

As with the Amiga's Workbench and CLI, the languages included with the machine are highly visible to both Amiga owners and potential owners alike. They serve as an indicator to many, of the useability and power of the machine, rightly or wrongly.

Should we have to apologise for any aspect of the Amiga when showing it to someone? I don't think so, and it's only a matter of having something done about anything that is embarrassing to show. Certainly the included language, AmigaBASIC, fits within the description of being an embarrassment.

Providing a decent BASIC implementation would open up the machine to those who are casual programmers. People coming to the Amiga from 8-bit home computer backgrounds would immediately feel comfortable with the programming tool provided, and would, in all likelihood, make significant use of the package, perhaps going on to programming in other languages in time.

The question that comes to mind is: 'Are there any other candidates for an included language?'

The answer to that is a definite yes. ARexx fans will tell you that ARexx serves them as well or better than any BASIC possibly could.

COMAL is another alternative, though as yet unreleased. The problem with these, of course, is a lack of familiarity on the part of the casual programmers. COMAL was available for all previous Commodore computers, and for MSDOS computers, but the number of owners using it were few in comparison with BASIC.

ARexx (in the form of Personal REXX), is available on the MSDOS machines, but again, there are very few actually using it.

While both of these languages are quite easy to learn, they are languages that still have to be learned, while most former owners of home computers are at least passingly familiar with BASIC.

It is my feeling that Commodore should approach a company like HiSoft with a view to licensing their package for inclusion with every Amiga. Amiga purchasers would love them for it.

What say you, Commodore?
Amiga Dispatches

A late Spring-Clean ... and fixing the BridgeBoard

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 3B2 computers. In his spare time, Don is an assistant Sysop on CompuServe's Amiga forums.

It’s odd how discussions about hypothetical situations suddenly become real. I’d had a brief discussion on CompuServe about one of the so called 'Murphy’s Laws'.

It started when someone said they had fried the $9,000 print-head of a particular brand of thermal printer. He had accidentally turned the power off, and then quickly turned it back on. Bye-bye print-head! This particular printer was manufactured by a company known for rock-solid, heavy-duty equipment so it was quite surprising that this type of failure had occurred.

The discussion led to how to build latch circuits and time delays to prevent this type of occurrence, and how cheaply one of those circuits could be built (especially when compared to the price of the print-head). That, of course, lead to my repeating one of Murphy’s Laws:

The cost of the equipment is inversely proportional to the cost of the part that will fail and destroy the equipment.

In other words, a $5.00 part will take out a $1,000 piece of equipment, but a $1.00 part will take out a $50,000 piece of equipment.

So, what’s all this got to do with the Amiga? Well, my friends, read on and you’ll find out.

How dirty is your computer?

I’ve had my A2000 for about a year and a half now, and I use it four or five hours a day. My computer room is quite small and in the basement, with no window or source of outside air. The basement doesn’t have any heat outlets in it, nor does my room. Like most basements, it doesn’t get out and out cold down there, but it does get a bit too cool to work comfortably. Because of that, I keep the room closed up and when I’m working, use a small electric heater to make it comfortable.

The room being closed doesn’t present a big problem, except for the fact that I smoke. I’ve tried various solutions to catch the smoke, and the obvious solution is to quit smoking but short of that, I’ve never found anything that’ll keep the air completely clean. This means that every year or two, I clean my computer to get rid of the built up film and dust.

A few nights ago seemed as good a time as any, especially since my hard-drive had developed a little bit of a squeal. The squeal I was hearing was typical of the sound caused by the static eliminator block on the spindle starting to rattle in the holder. It doesn’t mean anything is wrong, it’s just quite annoying.

Before you think that I got what I deserve because I smoke, let me say that I’ve had hard-drives that are brand new do this right out of the box. I’ve had drives that are kept in computer rooms, where the air is well filtered and conditioned and smoking isn’t permitted, also start to squeal. I’m sure my smoking doesn’t help, but it doesn’t cause the noise either, it just happens to some drives.

If you don’t know what a static eliminator block is, it’s a small conductive block held against the end of the spindle by a piece of spring copper that’s soldered to the ground plane of the drive. It prevents static build up on the drive platters and that’s a good idea, so you don’t want to remove them as I’ve seen folks do.

Anyhow, hard-drive and smoking aside, it’s a good idea to keep things clean as far as computers are concerned so I set to the task.

A word of caution is appropriate here. If you aren’t familiar with electronic equipment, you may not want to do this yourself. It is possible to ruin chips with static, you could knock parts out of alignment, use the wrong type of cleaner or lots of other things that can harm your computer. If you’re not sure of what you’re doing, please take the computer to a dealer and have their technician do the cleaning.

Of course they’ll charge you for it, but in the long run it’ll be worth the peace of mind knowing the job was done correctly.
The very first thing to do if you have a hard-drive is **back up the data and park the heads!** No matter how careful you are something can go wrong (isn't that also one of Murphy's laws?) and you want to make sure you can restore things if necessary.

After that, the first step in the actual cleaning is to disconnect the power cable, and all the other cables attached to the computer. Obviously we don't want to get a shock or have the computer turned on while we're cleaning it.

Next, we'll take the cover off the computer. Yuch! Look at all that dust build up by the vents, the mouse and keyboard connectors and any other opening in the front. Two thousand hours of drawing air through the computer will do that. It really doesn't matter how clean your room is, some dust will get sucked in and will accumulate around the front of the computer.

You don't want to take an ordinary vacuum cleaner and try to suck up that dust. Vacuum cleaners can generate a lot of static! Those little hand-held, so called 'computer vacuum cleaners' work okay, but you've got to make a very slight modification first - get the little bag damp so the dust doesn't go right through it.

Another way to get the dust off is to use a can of compressed air. You can get them at almost any camera store but **read the directions** on the can. Many of those cans of compressed air get extremely cold if used for more than a second or so at a time. Blow the dust away from the computer. I use a small paint brush to help clean things up, ... .

Now the hard-drives

I pulled all the cables to the hard-drive and floppy drives. A little trick that may help you is to mark the cables with where they go and which end is up. The drive cables (the ribbon style) usually have room on the top of the connectors to put a small strip of masking tape and you can use that to write on. I use a small paint brush to help clean things up, ...

... and then the floppies

On my 3.5" floppy (Matsushita), there are two very small screws holding a shield in place. By removing the screws and lifting off the shield, you can get to the heads quite easily. This also makes it easy to brush out any dust accumulated inside the drive. I then spray the same cleaner used on the hard-drive onto a cotton bud and carefully wipe the heads (there are two, one on top and one on bottom) side to side. Don't rub front to back, that way you don't have to worry about knocking the heads out of alignment. The chances of doing that are slim, but why take any chances? Side to side works fine and is safe.

The extra step I take is to place a tiny drop of light oil on the block. The operative word here is tiny. I've got a small oiler and put a single drop on the block and then immediately dab it with a paper towel to soak up the excess. This procedure has worked fine for me, and quieted down drives that nothing else would, but as they say in the car business ... your mileage may vary, so again, use it at your own risk. I went through the same procedure with the hard-drive I've got attached to the Bridge side.

Turning the hard-drive over, on the controller board you'll find a small copper plate over a hole in the circuit board. This plate will have a little (usually black) block on it that's held against the spindle. It should be fairly easy to identify. Once found, I clean it with some electronics cleaner. I use Tandy's Zero Residue Cleaner. The brand doesn't matter, just make sure the cleaner you use leaves no residue and is designed for electronic parts. **Never** use a water-based cleaner, nor any cleaner not designed for electronic equipment. I also spray the circuit board to clean off any built-up film. It doesn't take much spray - don't let it run onto the drive mechanism itself.

While that's usually enough to clean it and quiet things down, I go a step further - this next step is **not** recommended by hard-drive manufacturers. I've used it for years and not had any problems, but that doesn't mean you won't have problems - so if you are going to try it, remember you do so at your own risk!

The very first thing to do if you have a hard-drive is **back up the data and park the heads!** No matter how careful you are something can go wrong (isn't that also one of Murphy's laws?) and you want to make sure you can restore things if necessary.

After that, the first step in the actual cleaning is to disconnect the power cable, and all the other cables attached to the computer. Obviously we don't want to get a shock or have the computer turned on while we're cleaning it.

Next, we'll take the cover off the computer. Yuch! Look at all that dust build up by the vents, the mouse and keyboard connectors and any other opening in the front. Two thousand hours of drawing air through the computer will do that. It really doesn't matter how clean your room is, some dust will get sucked in and will accumulate around the front of the computer.

You don't want to take an ordinary vacuum cleaner and try to suck up that dust. Vacuum cleaners can generate a lot of static! Those little hand-held, so called 'computer vacuum cleaners' work okay, but you've got to make a very slight modification first - get the little bag damp so the dust doesn't go right through it.

Another way to get the dust off is to use a can of compressed air. You can get them at almost any camera store but **read the directions** on the can. Many of those cans of compressed air get extremely cold if used for more than a second or so at a time. Blow the dust away from the computer. I use a small paint brush to help clean things up, ... .

Now the hard-drives

I pulled all the cables to the hard-drive and floppy drives. A little trick that may help you is to mark the cables with where they go and which end is up. The drive cables (the ribbon style) usually have room on the top of the connectors to put a small strip of masking tape and you can use that to write on. I use a small paint brush to help clean things up, ...

... and then the floppies

On my 3.5" floppy (Matsushita), there are two very small screws holding a shield in place. By removing the screws and lifting off the shield, you can get to the heads quite easily. This also makes it easy to brush out any dust accumulated inside the drive. I then spray the same cleaner used on the hard-drive onto a cotton bud and carefully wipe the heads (there are two, one on top and one on bottom) side to side. Don't rub front to back, that way you don't have to worry about knocking the heads out of alignment. The chances of doing that are slim, but why take any chances? Side to side works fine and is safe.

The extra step I take is to place a tiny drop of light oil on the block. The operative word here is tiny. I've got a small oiler and put a single drop on the block and then immediately dab it with a paper towel to soak up the excess. This procedure has worked fine for me, and quieted down drives that nothing else would, but as they say in the car business ... your mileage may vary, so again, use it at your own risk. I went through the same procedure with the hard-drive I've got attached to the Bridge side.

Turning the hard-drive over, on the controller board you'll find a small copper plate over a hole in the circuit board. This plate will have a little (usually black) block on it that's held against the spindle. It should be fairly easy to identify. Once found, I clean it with some electronics cleaner. I use Tandy's Zero Residue Cleaner. The brand doesn't matter, just make sure the cleaner you use leaves no residue and is designed for electronic parts. **Never** use a water-based cleaner, nor any cleaner not designed for electronic equipment. I also spray the circuit board to clean off any built-up film. It doesn't take much spray - don't let it run onto the drive mechanism itself.

While that's usually enough to clean it and quiet things down, I go a step further - this next step is **not** recommended by hard-drive manufacturers. I've used it for years and not had any problems, but that doesn't mean you won't have problems - so if you are going to try it, remember you do so at your own risk!
... now, the circuit boards

My next-to-last step inside the computer is to pull all the expansion cards and clean their contacts. I do this with a coarse paper towel, sprayed with the Zero Residue Cleaner and wiping the towel across the contacts. I don’t do much else with them except to brush off any dust on the boards.

That leaves me with a bare mother-board. With it open like paper towel, sprayed with the Zero Residue Cleaner and expansion cards and clean their contacts. I do this with a coarse this, it’s real easy to brush off any accumulated dust on the board itself. I go through and check all the socketed chips to make sure that they are properly seated. Just a quick push down on them is all that’s needed. If they’re loose, you’ll feel them move down. If not, you don’t want to push too hard as you could damage the mother-board.

... putting it back together

Re-mount the 5.25” drive in its bay and then re-mount the 3.5” hard-drive onto the mount plate. Make sure you’ve got it seated far enough back to clear the front cover.

Next goes the drive mount plate, with the drives attached, onto the power supply-drive bracket and then the whole works back into the computer. Make sure the tabs on the bracket are lined up properly and then put the six screws back in. On my A2000, the two screws in the front didn’t have washers, while the four in the back did. I don’t know if this is typical, but if it is, you now know which have washers and which don’t.

Now, put the cards back in their slots. I can’t stress how important it is to get the cards fully seated; a card like the BridgeBoard will just give you problems if not properly seated. Also, remember that the bracket on the back can become hung up on the brace under the mother-board. As I stated in one of my earlier columns, I bend the bottom of the bracket on the card out a bit and that makes installation of the cards much easier.

The next step is to hook the cables back up. Make sure the cables are all attached to the various drives properly. If you took my advice about marking them, that should be quite easy. The power cable to the mother-board will snap in when it’s properly seated. The power cables for the drives are ‘keyed’ by their shape for the large plugs, but the small floppy plugs are only ‘keyed’ for up/down but not side to side, so be careful.

Okay, check your work. Does everything look right? If so, go ahead and attach the external cables to the rear with the exception of the power cable. The power cable is the very last thing you’ll hook up. You can leave the cover off for now, just in case something went wrong.

... so, to the keyboard

You’ve got to be very careful here. You can clean the keycaps with them either in place, or removed, but I would strongly suggest you only remove them if you’re having a problem with one of the keys sticking, or not working properly. If you clean them while they’re on the keyboard, use a damp rag with some cleaner on it. You do not want the rag to be dripping wet because you don’t want to get any liquid into the keyboard itself. Again, use a household cleaner with a cotton bud to get into the tight areas.

If you have sticky keys, or keys that don’t respond to a keystroke, you’ll have to remove the keycaps. It’s very important to be careful because you can ruin a keyboard if you don’t pull the keycaps off properly! The keycaps are held on by latches on both sides of the cap. You can buy a special tool from many computer stores for keycap removal. An alternative is to use something like a small button hook and to hook the keycap from one side and gently pull up and to the side at the same time. The cap will suddenly pop off. The larger keycaps, such as the space bar, are also held on by a guide wire. They can be real tricky to put back, so it’s best to leave these alone.

When you get the keycap off, what you see depends on the keyboard manufacturer. My particular keyboard is made by HiTek and has small leaf switches that are held open by the slider mechanism. Pushing down on the slider closes the switch and letting it go opens it back up. The leaf switches are very delicate so stay away from them! A quick spray with the Zero Residue Cleaner should take care of problems with a sticky key or one that doesn’t respond when you strike it.

If you accidentally pull the whole slider mechanism off, don’t panic, they’re easy enough to put back on if you’re careful. As I said, the leaf switches are held apart by the slider mechanism and if the slider comes off, you can’t just push it back in. Take something like a toothpick and put it between the leaves and then lift up on the toothpick to open the switch. Push the slider onto the spring and slowly push it over the switch, making sure you don’t catch it on the switch leaves. Once the slider is over the switch, it’ll snap into place on the backing plate.

Putting the keycaps back is easy. Just put them over the key oriented properly and push down on them, they’ll snap into place.

... and the mouse takes a bath

The only thing I do to the mouse is to clean the ball and rollers inside the mouse. If you turn the mouse over, you’ll see the access plate for the ball. It rotates to come off, there’s usually an arrow indicating how to turn it and your manual also explains how to remove the ball. The ball is easy to clean, a little soap and water and it’s done. Just make sure it’s completely dry before you put it back in the mouse. While it’s out, check the rollers inside the mouse. They should be clean and shiny.

Now, there is no little rubber band in the middle of the rollers - but if you see something that looks like a piece of rubber on the rollers, it’s dirt and should be cleaned. If you’ve got a magnifying glass, take a look at the ‘rubber band’ and then slowly and gently scrape it off. Use a wooden toothpick to do the
scraping. Once the ball and rollers are clean, drop the ball back in place and put the access cover back on.

... and finally, the monitor

Go ahead and plug the mouse and keyboard back in to the computer. That leaves only the monitor screen left to be cleaned. There are things you must never do around a monitor! A monitor (colour or monochrome) has high voltage, up to 30,000 volts, driving the tube and that can KILL you!

Never clean a monitor while it is turned on. Never use a dripping wet rag to clean a monitor. Never spray the screen, spray a rag with the cleaner and then use the rag to clean the screen. Never take the covers off the monitor or poke around inside the monitor - leave things like that to a qualified repair person. They have the proper tools and knowledge to work safely around those voltages.

To clean the monitor screen, spray a rag with a glass or household cleaner, and then wipe the screen until you’ve got it clean. Again, please be careful not to get any liquid inside the monitor. The screen will clean up quite nicely with small amounts of cleaner.

Okay, now everything is clean! Make sure it’s all dry and hook the monitor back up to the computer. Check all the connections. If you’ve still got the covers off the computer, re-check all the cables. Finally, plug the power cord back into the computer and turn it on. If anything abnormal happens, turn the computer off immediately!

Murphy strikes again

When I did this ... aargh!!! The computer didn’t turn on! The power LED didn’t light, the power-supply fan didn’t come on, the drives didn’t spin up! What had I done wrong?

I looked at all the connectors but didn’t see anything immediately obvious. The plug was plugged into the wall socket and the house circuit was okay.

Well, I pulled the power connector to the internal drives and tried to turn the power on again. The fan spun, and the computer started to boot. With no power to the drives, it couldn’t do anything, but I was encouraged. I shut the power off and then plugged the hard-drive back in. I turned it on and everything spun up properly, but the floppy still wasn’t powered, so I still couldn’t boot.

I reached behind the floppy and tried to insert the power connector into the floppy’s socket and realised that it would fit in several positions! The way my computer room is set up, it’s not easy to get behind the computer to look at the floppy power socket and it’s not visible from the top. I grabbed a mirror and looked at the connection and that’s when I found what I had done wrong. Without side shields on the socket, there is nothing to prevent the connector-socket combination from being put on offset side-to-side! You can’t put them together upside down, but you can sure put them on offset by a pin or two! That meant that I was running either the 12v or 5v feed directly to ground! No wonder the computer wouldn’t start. I installed the connector properly and turned the computer back on ... it booted and ran just fine.

And that’s where Murphy’s law comes in. For lack of about a tenth of a cent’s worth of plastic to mould-in side panels on the socket, I could have ruined a multi-thousand dollar computer. This isn’t Commodore’s fault, they buy the drives from various manufacturers and that’s the way the drives come. It’s the fault of the engineer who designed the ‘standard’ plug and socket combination used on those drives. It’s also my fault for not grabbing that mirror earlier and looking at what I was doing. I put the connector on by feel and thought I had it right ... obviously I didn’t.

So take a lesson from Murphy and me. No matter how many computers you’ve taken apart, and no matter how familiar you are with electronics equipment ... take the time to make sure you’ve done everything right. As Murphy’s first law says, ‘If anything can go wrong, it will!’

The cleaning was worth it. It only took a few hours of my time, and the hard-drives no longer squeal. The keyboard feels crisp again and the monitor looks much brighter with that layer of dirt off the screen. I don’t have to worry about the accumulation of dust inside the floppy drives and overall, the computer just looks better ... just like when it was new.

Bridgeboard repairs

I never did want to become a ‘guru’ on the BridgeBoard, but it seems that problems and/or new things keep cropping up related to the Bridge all the time ... so here’s some more.

My BridgeBoard died a week ago. It had been fairly flaky for a long time. It seemed to have a mind of its own when it came to booting. Sometimes it would boot just fine, and sometimes it would hang or Guru the machine. It also seemed extremely sensitive to how exactly it was positioned in the slot. Any deviation from ‘just right’ and it wouldn’t work.

Well, last week, it decided it was just going to quit ... it wouldn’t work no matter what I did. It wouldn’t access the drives, and the BindDrivers command would simply hang, either locking the Amiga or giving me a Guru.

For those of you think I must have done something dreadful while cleaning the machine ... nope, the card seemed to work fine for a while after that. In fact, it had seemed to work better than before, no problems at all since the cleaning ... until this.

In any event, I tried all my little tricks to get things working again, such as re-seatting the chips, boards, re-loading the software, and so on. That didn’t work, so off to the dealer’s for a repair.
This is where things got interesting. The service technician told me that CBM supplied him with a diagnostic disk for the Bridge, but that to use the disk the Bridge had to be booted. Talk about Catch-22. You can't diagnose problems unless you can boot the Bridge, but if the Bridge is broken, it won't boot. Commodore's answer to that problem is to send the Bridge in for a swap ($2351)

Now Roger (the service tech) and I are smart enough to know that CBM has to have some form of test jig for the Bridge. He made several calls to CBM and found out that in fact, there is such a test jig and program available in Europe but not here in the US. It seems the folks in Europe won't stand for board swaps ... they want chip-level diagnostics, so CBM in Europe was able to provide them with what they wanted. The Commodore folks here in the US don't feel the same way, they figure we'll stand for board swaps rather than provide board level diagnostics for the repair shops.

The Commodore folks here said that the board level stuff wasn't 'just right' (it had some bugs) but that at some time in the future, they would release the board level diagnostics (and test jigs where necessary) when those programs or hardware are 'just right'.

Let me digress from the specific subject of the Bridge Board repair to repairs in general for a minute. In some cases, a board swap is the right way to go; in some cases, it isn't. Service tech time runs at about $40 to $60 per hour, depending on where you are and how good the tech is. Some problems, in particular intermittent problems, are extremely hard to diagnose. Most of the chips in the computer are soldered in place. Soldering the chips in place lowers production costs and increases overall system reliability. You don't have to worry about loose chips, but you do have to worry about repair costs.

An example of the repair problems (and the expense) that this can cause is in a situation where you attempt to trace a signal. The signal is fine going into chip A, but disappears on the output of chip A as it heads off towards chip B. That could mean that chip A is bad, having no output, or that chip B is bad, shorting the input. With socketed chips, you simply pull (or replace) chip B, and if the signal returns, B is bad; if it doesn't, A is bad.

With chips soldered in place, it's not so easy. With multi-layered boards, you've got to be very careful removing a chip. It's quite easy to destroy a multi-layered board with improper soldering techniques. It's also not unusual to destroy the chip being removed. With a $3 chip, that's no big deal, with a $50 or more chip, it is a big deal.

So, the tech has to make an educated guess as to whether A or B is bad and replace it. If the tech is wrong, then they've got to replace the other chip also. All that takes time and money plus the parts cost, which can be rather high. Cases like that, where you can't quickly isolate the problem and fix it, a board swap is probably a good idea. It's less expensive for the customer in the long run. Not only that, but if the tech takes three or four hours per machine isolating a problem, they'd pretty soon have a pile of machines awaiting repair. Of course, that would also upset the customers having to wait a week or two before the tech could even look at their machine.

However, there are cases where a board swap is ridiculous. One of the members on CompuServe reported that his A1000 went bad. He took it into the shop and they told him that he needed a new mother-board. That cost him $180 plus the labour to replace it. When he asked what was wrong with his mother-board, he was told, 'One of the memory chips was bad.' Now, if that wasn't just a standard line that shop feeds customers when they ask what was wrong, then spending $200 or more, because a $10 chip went bad is garbage! If they were able to figure out that it was a single memory chip, then they should have been able to isolate that chip and replace it for one heck of a lot less than $200.

But, because Commodore won't supply the proper diagnostics, many shops don't hire fully qualified techs. They hire folks who have minimal diagnostic skills but are able to swap socketed chips or whole boards. They have turned from repair shops into 'board swap' shops. Their so-called techs, do minimal checks of the chips they can replace and if those aren't bad they send the board in for a swap.

If Commodore ever gets their diagnostic programs 'just right', then perhaps we can have real repair shops again. Shops where the tech would know which problems really needed a full board swap, and which problems could be fixed fairly easily in the shop.

Right now, I think that Commodore is doing the Amiga community a real dis-service by not providing the techs with the proper tools. Even tools that aren't fully debugged would be better than no tools at all in my opinion. As long as the limits of the current diagnostic software is known, then a qualified tech could work around those limits.

Anyway, those tools aren't currently available here in the US, so back to my BridgeBoard problem.

Before we decided to send my Bridge in for a swap, Roger decided to try to figure out the problem. He had a working Bridge in his shop and from talking with Commodore had an idea where the problem might be. At the very least, he could try swapping some of the chips that are socketed with those on the working Bridge to see if he could isolate the problem to a specific chip.

He replaced the chips one at a time, until suddenly the board acted like it was going to work. As he pulled the chips, he bent the pins of the chip carrier out a bit to ensure good contact when the chip was re-inserted (these were the large square chip carriers). Of course, along the way, he also had to re-seat the board a few times because no matter what, it's still got to be seated right and it's not the easiest board to seat.
What was happening now was that it would access the drive when it hit the BindDrivers command, but the drive wouldn’t reset. Roger said that he’d seen that problem on the earlier 5.25" drives that Commodore supplied with the Bridge. Sometimes, they seem to simply hang on track 0, they won’t seek and won’t reset, even with a full power off/on cycle. He had a real simple fix for that. You power off and move the head on the drive up a bit so it’s off it’s parked position. That seems to wake up the drive next time you turn it on. By the way, this is the 5.25" floppy drive, do not move the heads on a hard-drive!

Well, darn if that didn’t do it. Turning the Amiga back on, the Bridge booted just fine now that the drive remembered it was alive. Back to Catch-22, there was no need to run diagnostics now, the board was working fine. Well, now we had to put the original chips back in one at a time to see when it broke again ... and hopefully that was the chip that needed replacement.

Yup, you guessed it, all the original chips went back in and the board continued to work just fine. Apparently the problem was a bent pin in one of the chip carriers (probably the Bus Translator chip). It just wasn’t making good contact and normal oxidation of the metal finally caused it to stop making electrical contact. An hour’s worth of labour, and my Bridge was fixed. I brought it home, put it back in my Amiga and it’s been working just fine now. In fact, better than ever. No more positioning tricks. It slipped in and started working right away although I did have to do the move-the-head trick on my 5.25" drive the first time.

Since then though, it’s been fine. This head locking wasn’t the original problem. Even though the heads were locked, the LED on the drive would come on, and the bridge would boot to the point where it attempted to look for the DOS disk in drive A:. When I first brought my Bridge in, it wouldn’t even pass BindDrivers and the drive LED would not come on at all. I suspect that when the Bridge failed, it somehow or other caused the heads to lock in position. I don’t know if that’s really possible, but it’s interesting that it occurred on two machines when using my Bridge while it was defective.

I ended up paying about a fifth of what a board swap would have cost me. The moral of this story is that, if your Amiga ever breaks, try to find a shop where they’ll at least be willing to try a few things before they decide you need to spend several hundred dollars on a board swap. A competent tech can do some fairly quick tests on his own to see if the problem can be easily isolated and fixed. A good tech can usually tell, fairly quickly, if the cheapest route for you is a board swap or shop repair. A good tech will also do the little things like re-setting the pins on a chip carrier just to make sure the problem is the chip and not the carrier (as my case turned out to be).

It might be useful for you to go down to your dealer’s and have a chat with the tech. Even if your Amiga is working fine, you’ll at least get some idea of how that particular shop works. If they are just a board swap shop, you might want to look around for another repair shop. Hopefully, your Amiga will never break, but if it does, you might save yourself some time and money if you know of a good repair shop beforehand. And while you’re looking around, you might want to see if the tech can try to convince Commodore to release what diagnostic software they have now.

New Toys

While I was at my dealer’s, I got a chance to look at the new A2286 AT-compatible Bridge Board. They had received three of them the day before, the first three in Denver.

It’s a nice unit, but rather pricey at a list price of $1499. The A2286 is an 8MHz, 1 Mbyte RAM, AT-compatible clone on a card/daughter-board combination. It takes up two Amiga slots, one of which has to be a Bridge slot. That means, if you put it in the left Bridge slot, you’ll no longer have a fully AT compatible slot left. However, if you wish, you can have the extra two 36-pin card edge connectors installed to make all the MS-DOS slots fully AT compatible. The holes, along with the circuit traces are already on the A2000’s mother-board. The connectors simply need to be soldered in place. If you place the A2286 in the right Bridge slot, you’ll use up one of the Amiga only slots also; however, you have one fully AT compatible available with no mother-board modification needed.

The A2286 comes with a 1.2 Mbyte 5.25" floppy, MS-DOS 3.3 and the new JANUS release 2.0 software and supports MDA and CGA video modes through the Amiga’s monitor (just like the A2088 Bridge). There is also a socket on the board for a 80287 math chip if you wish to add one.

The board looks well built (no last-minute wire add-ons) but there’s one glaring omission I see with the board. It has no external drive connector like the A2088 has! If you want to add a second floppy, you’ll have to provide a drive, power supply and case and then run the ribbon cables into the back of the A2000 through an open slot hole. Finding an open slot hole is no problem, you can remove the cover from the slot to the right of where you’ve added the A2286’s main board. That slot is taken by the A2286’s daughter board anyway so can’t be used for an expansion board anyway. But to be honest, that’s going to look sloppy. Also, since it doesn’t have the db23 connector, you can’t directly use an A1020 (Commodore’s 5.25" Amiga add-on drive) with the A2286.

Unfortunately, my dealer hadn’t installed the card in a unit yet for demo purposes so I couldn’t test its functionality. However, I could test the new JANUS release 2.0 software, because the same software works with the A2088 and the SideCar. All the software is completely changed and some new things have been added.

You can now use the A2000’s clock to set the date/time on the MS-DOS side. You can use the Amiga’s mouse as a MicroSoft bus mouse. You can now autoboot the Bridge from an AmigaDos file used as a real (as opposed to virtual) drive. In fact, this drive is so ‘real’ to the Bridge side, you need to use FDISK to partition it and FORMAT to format it before you can access it.
Not only that, but when you create it, using the Amiga program MAKBAB, it gets created properly full-sized. LINK (which is still available) wouldn't create its virtual drive full-sized, so you had to fill it first to prevent errors if you forgot to unlink the drive (see Amiga Dispatches in TransAmi, volume 1, issue 1 for more information on this subject).

AREAD and AWRITE have additional options beyond the /b switch, and this time, they are documented. By default, AREAD and AWRITE do both character conversion and line-feed to carriage-return/line-feed conversions during their operation. The new switches allow you to turn one or the other conversion off.

And after all that mess we went through a while ago getting FFS to run on a JANUS drive, DMOUNT now fully supports FFS on a JANUS drive. Also included is a new Amiga FORMAT command that works on a JANUS drive under the OPS or FFS. By the way, one more little quirk popped up using the old JANUS software and FFS on a JANUS drive. A few folks tried it using the first partition on the MS-DOS hard-drive as the JANUS drive and it didn't work at all. They moved the JANUS slice to the second partition, and it worked fine.

One other thing that's nice about the new software is that if your Bridge fails to boot, you'll no longer get a 'Task Held' requester and then a Guru when you cancel the requester.

Some of the directories on the Amiga side have also been moved around, and you can't mix versions of the software. So if you intend on upgrading, make sure you replace all the necessary files, and that they go in the proper place. Also, make sure you run PCPrefs, and set the RAM location for the JANUS load properly before you try to use the Bridge. For the SideCar and A2088, the proper RAM location is $F000 while the A2286 requires it's RAM location to be $D000.

At this point, Commodore has not come up with a policy on how to upgrade from the original JANUS software to the new release 2.0 software. It's unknown if dealers will be able to make a copy of the disks for folks who have the current software, or if they'll have to buy it as a separate package. If you want to upgrade, get hold of your dealer ... by the time you read this a policy may have been set.

More toys coming

AMIX (Amiga UNIX (tm)) and the A2500ux have been described in a 'Technology Preview' bulletin from Commodore. A Technology Preview is not an announcement of availability of the product, nor is it the final specification of the product. It is merely an announcement of a product as it currently stands and is expected to become available sometime in the future. The bulletin states the following about AMIX and the A2500ux:

AMIX will be fully compatible with AT&T UNIX System V, Release 3.1. It will be able to address 1 Gigabyte of virtual address space, with paged memory management. Software support for multiple users, multiple video screens, streams and a remote networked filesystem (RFS) is included. A proprietary Windowing interface is included.

Software included with AMIX will be an optimising C compiler with standard development tools such as cc, sdb, make, yacc and lint. The standard text utilities will also be included such as email, uucp, nroff, vi, man, spell, and so on.

The A2500ux with a beta version of AMIX is available now to qualified members of the Amiga Commercial Developer Support Program.

Another Technology Preview bulletin talks about the High Resolution Colour Graphics Card. This card will support resolutions up to 1024x1024 with 8 bit-planes providing 256 colours (out of a palette of 16 million colours) plus two overlay bit-planes providing an additional three colours simultaneously. A TI TMS34010 high speed graphics processor will execute graphics at up to 6 million instructions per second. A standard TIGA graphics library provides a library of high speed graphics commands for the TMS34010.

The PVA2350 Professional Video Adapter is also described in a Technology Preview bulletin. The PVA2350 is a real-time frame grabber operating in real time at 1/60th of a second resolution. It can capture odd, even or both video fields in full colour and supports all Amiga graphics resolutions including HAM. The card also works as a digitizer and genlock with both RGB and composite outputs.

Finally, Commodore also released a Technology Preview bulletin on the A590 Hard Disk controller and expansion memory unit. The A590 is a complete 20 Mbyte hard-drive specifically designed for the A500. It will auto-config and auto-boot under v1.3 of the Amiga operating system. It operates as a DMA device, attaches to the side port of the A500 and has a Mac-style SCSI connector on the rear to allow additional drive expansion. The A590 contains sockets for up to 2 Megabytes of expansion RAM. While not included in the bulletin, it's my understanding that the A590 will also work on an A1000.

Some of these were officially announced as products (rather than Technology Previews) for the European markets at the recent CeBIT show held in Hanover, West Germany. That means they should also soon be available in the US market.

While I have griped above about some of Commodore's service policies, I've got nothing but good things to say about Commodore's Engineering Department. The above previews tell me that Commodore is working hard to make the Amiga a very serious contender in the computer marketplace. That can mean nothing but good for those of us who already own an Amiga.
BRIWALL AUSTRALIA

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>$159.00</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>Excellence</td>
<td>$359.00</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>$55.00</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>Assempro</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:

BRIWALL AUSTRALIA
PO BOX 9, RIVETT ACT 2611
TELEPHONE (062) 88 0131
FACSIMILE (062) 88 0337
Driving the Drives

Beware, dangerous driving can be hazardous to health!

by Dr. Rahman Haleem
Commodore (UK) Technical Support

There have been reports of some games not working on the A500.

The reason is that some software developers created their own FDD software interface. In writing their own loaders, there are five points that would cause loading problems:

- the step pulse has the wrong polarity
- the wrong step rate is used
- a software loop is used for timing
- no error checking is done
- the READY line for the external drive is screwed up

Disk access on the Amiga

1. When turning the motor on, wait for the READY signal to go low before reading or writing - however, steps are fine before then.

   Note: the READY signal is only valid when the motor signal is on. The READY line does not work the same for all drives 0 and 1 (or above), and in some instances, cannot be relied on at all.

2. When pulsing the STEP line, remember that STEP is defined as a short, low-going pulse. The recommended method is to pulse the STEP line LOW, then HIGH, and wait for the step delay period.

   Do not pulse LOW, wait the STEP period, then pulse HIGH, as this looks like a high-going pulse.

3. Do not reduce the STEP rate below the 3ms allowed by our specification. Some drives will die at faster step rates - there is one which fails at 2.8ms - you have been warned.

   When changing step direction, you also need to add the 15ms settling time to the 3ms step time. This means that you would need to wait a minimum of 18ms.

4. Do NOT use software loops for timing during loading (or in your programs). Software timing loops cause problems if faster processors are used. Instead, use the timers in the 8520s for any timings that you don't want to speed up when using different processors.

5. Do NOT remove all error checking. The system should be able to report a bad disk, rather than just fail if there is an error during the load process.

6. Do NOT use self-modifying code in your loader. Some processors (such as the 68020) have instruction caching, and self-modifying code will not work!

7. We at Commodore do not guarantee that our drives will have more than the normal 80 tracks. If you use three additional tracks, we do guarantee that you will break.

8. After a disk write DMA has finished, a delay of 1.2ms is required before any other operations, such as drive select, step, head change and so on. Failure to wait out the delay may result in writing over, or erasing other data.

9. To determine if a disk is in the drive, look at the DISKCHANGE signal. If it is low, the disk has been removed (and possibly re-inserted again) since the last check. Step the head to reset the latch, and examine the current state.

10. Above all, do not make assumptions about the state of the drive. For example, if you depend on the drive motor being on, then turn it on!

[The above was received from Dr. Haleem on the day after this edition had been finalised! We thought it important that we should include the warnings - however, because it was received so late, it has not undergone our normal scrutiny. Therefore, please address any queries direct to Dr. Haleem at Commodore UK Ltd. - ED]
Transactor for the AMIGA

Access

The world of non-commercial software

by 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.

AmigaZOO 2.0
Shareware by J. Brian Waters

Programs like ARCHIVE, AmigaZoo, PAK, and I'm sure they will be others in the future, were written mostly for use in telecommunications. They make it easy to transfer a program (which may contain more than one file) via a modem. They also allow you to create a single file which can contain many other files and which is normally significantly smaller, usually 50 to 60 percent smaller, than the total size of the files contained within. This is done by various compression techniques. When transferring files to and from a commercial network or a long distance BBS, the smaller the file, the less time connected and the less charges you incur.

These programs are also used to reduce the storage size of information you may want to keep available to you. They are handy indeed, allowing you to store twice as much information on a disk as you could without them.

Very shortly after the Amiga was released, Raymond Brand ported the ARCHIVE utility (originally by System Enhancement Associates) to the Amiga. This became the de facto standard in librarian and file compression programs for the Amiga by default; there were no others at the time.

ARCHIVE (ARC) was perfect for Amiga's needs. It librated and compressed files, and was compatible with versions of ARC for other machines. You could download text files created on MS-DOS machines, STs, and anything else that had the utility available, then unARC and use them on the Amiga. Unfortunately, Amiga ARC has not kept up with the advances in the program in the MS-DOS world. There are more and more times now that I'll download an ARC file created on an MS-DOS machine and try to unARC on the Amiga just to be met with the 'You need a newer version of ARC' message. This is because newer compression techniques have been added to the MS-DOS version. The Amiga version has been dormant for a while.

Two other programs that provide the same functionality as ARC (plus more) are becoming popular. PAK and AmigaZoo (ZOO).

I'm not too crazy about PAK for reasons I've mentioned before, mostly because of poor compression. ZOO, on the other hand, is gaining in popularity for a variety of reasons.

AmigaZoo is compatible with the MS-DOS version. It appears to be well supported, maintained, and upgraded as the MS-DOS version is upgraded. Besides some of it's technical advantages over ARC, the biggest advantage, for me, is that it is being actively supported.

Unlike ARC, ZOO supports full-length AmigaDOS filenames and can archive and restore directory structures. This is a nice feature. Many times, using ARC, if you had a program that had names longer than the MS-DOS standard allowed, or the program required some of its files to be in sub-directories, you would have to write a script, or use another program the would create a script for you, and include that script in the ARCed file so those downloading the program could successfully set it up.

ZOO is friendlier to single disk drive users than ARC is. ZOO allows a single drive user to keep the program in the C: directory, yet when it is run it will wait for a carriage return before doing any disk io. This allows the user to change disks. He doesn't have to have ZOO, the ZOO file, and the resulting destination file(s) on the same disk. Some extracted files can be large and may not fit on a normal workbench boot disk.

ZOO has many more features than ARC, yet those features you are used to with ARC have the same command line arguments. Little relearning is required.

Since AmigaZoo 2.0 has been released, a few minor bugs have been found. These are being fixed shortly after they are reported. Recently a 2.01 patch file was released to fix a problem with the date stamp.
While Zoo is not as widely used as ARC, it is gaining popularity because of its features. Sometimes, depending on file type, ARC is more efficient in compression on the same file. I'm willing to accept this. The difference is usually small and Zoo really shines by allowing full AmigaDos pathnames and the archiving of sub-directories. Any loss in compression is usually more than made up for by not having to supply a script with ARC files to recreate the correct names and directory structures.

AmigaZoo is a copyrighted shareware program. To quote from the documentation:

'AmigaZoo is a user supported program. Zoo was written by Rahul Dhesi and modified to run on the Amiga by J. Brian Waters. With the consent of the author, I am requesting a $10-20 contribution from satisfied users. This contribution is voluntary and you need not feel any guilt if you do not contribute. Any contributions received will go toward one of three things: firstly, and most importantly, they will be used to acquire any tools which may appear on the market to help make AmigaZoo better, like more optimising C compilers or better debuggers; secondly, it will be used to help defray the costs of distributing Zoo; thirdly, if any money is still remaining it will be used to buy some shares of Commodore stock.'

AmigaZoo is available in CompuServe's AmigaTech Library 4 as Zoo.ARC. AmigaZoo's author, J. Brian Waters, may be contacted by sending CompuServe Email to 73127,2124.

setCPU 1.4
Public Domain software by Dave Haynie

Are you one of the lucky people to have the A2620? If so, you need SetCPU!

SetCPU is intended to allow you to detect and modify various parameters related to 32 bit CPUs (68020 and 68030) in Amiga computers.

When run with no parameters the program will give you information about your system. Unfortunately, on my system, I get:

2> setcpu
SYSTEM: 68000

On a system with the A2620 you will get:

SYSTEM: 68020 68851 FASTROM (INST, CACHE)

What this tells us is that you are running with a 68020 CPU, 68851 math co-processor, 68851 MMU, and have moved the ROM routines into 32 bit RAM.

ROM routines in 32 bit RAM? Yes! That is what this program is really all about. You will see a dramatic speed improvement in all system calls by having the ROM located in 32 bit RAM. This feature isn't necessarily limited to the A2620. In theory it should work with 3rd party boards with MMUs. It hasn't been tested by the author on boards other than the A2620. It will have to be modified a bit to work properly with memory outside of the 68000 compatible 24 bit address space.

The syntax of the program is....

setCPU [INST|DATA] [[NO]CACHE] [[NO]BURST]
[[NO]FASTROM [TRAP]] [[NONMUTEST]]
[CHECK 680x80 6851 6881x [MMU|FP]]

The various parameters, as explained in the program's documentation, are as follows:

[NO]CACHE: This command will switch on or off 68020 and 68030 caches. If not qualified, it will act on both instruction and data caches of the 68030.

[NO]BURST: This command will switch on or off the burst cache line fill request of the 68030. If not qualified, it will act on both instruction and data caches.

INST: This precedes a CACHE or BURST operation to restrict it's application to the instruction cache only.

DATA: This precedes a CACHE or BURST operation to restrict it's application to the data cache only.

[NO]FASTROM [TRAP]: This activates the PASTROM translation on or off an MMU-equipped system. When switching on,
it first allocates 256K of memory for the ROM image, then 512 bytes of memory for the MMU table. It copies the ROM into the image area, then applies the translation by pointing the MMU at the table and activating it. The TRAP option causes the MMU to look at all 32 bits of address; access to any memory outside of the 24 bit space will result in an exception, which if unhandled, results in a Guru 2. When TRAP is not specified, the MMU is only looking at 24 bits of address space. This is the mode you'd normally run in. Under V1.3 and earlier releases, a DOS bug can cause invalid accesses, which cause the exception, when running the Endcli or Newcli/NewShell programs; running untrapped will avoid gurus with these commands.

It is still possible to run into an exception when running without the TRAP option. One easy way is writing to the ROM image, which is write protected by the MMU. The NOFASTROM option will switch off the MMU and reclaim the memory used for the ROM image and MMU table. Note that this is achieved by locating these items via the appropriate MMU registers. If any other program set up the MMU for something, this could be a very bad thing to do. In general, until there's some level of OS support for the MMU in Amiga systems, you're really safe using only one MMU tool at a time.

NOMMUTEST: This option won't normally be necessary. However, some 68020 boards out there apparently have hardware bugs of some sort that make testing for the MMU impossible, at least as near as I can tell. If your 68020 board freezes, or crashes with some kind of co-processor protocol violation Guru (probably numbers 50D or 50E), you have a problem with your 68020 board. The most common bug is the 68020 board hardware failing to fully decode CPU space, such that the FPU in the system also responds to the MMU's address space.

CHECK: This option lets you check for the existence of a particular CPU system component in a script. It works like this:

```bash
SetCPU CHECK 68020

If WARN
  echo "No 68020 here!"
Else
  echo "Sho nuff got a 68020 here!"
Endif
```

The arguments to CHECK can be any of:

- 68000
- 68010
- 68020
- 68030
- 68851
- 68881
- 68882
- FPU (Matches 68881 or 68882)
- MMU (Matches 68851 or 68030)

SetCPU, C source and executable, has been placed in the public domain by Mr. Haynie and SetCPU.ARC is available in CompuServe's AmigaTech Library 14.

PathAss is shareware software by Anders Lindgren

PathAss is a nice contribution to the Amiga shareware library by Anders Lindgren of Sweden.

What PathAss does is allow you to assign devices across directories or even disks. How often have you wished you could use:

```
ASSIGN Fonts: DFO:font AND DF1:FONTS
```

With PathAss you can! To use it, you type something like:

```
PathAss DEVS: vd0:devs,df0:devs,df1:devs
```

The order of the directories is significant, they are searched from first to last. You can easily list all current assignments or delete any or all assignments.

Another feature of the program, which many might consider not such a good thing, is that the assignments are not volume relative. In other words, if you use

```
PathAss FONTS: RAM:,DF0:/fonts
```

PathAss will look in RAM, then in the root directory of the disk in DFO, whether that was the disk when you made the assignment or not. It looks at the device, not the volume name. This could be handy if you have many disks with fonts on them for which you would like to have easy access.

PathAss is shareware. The shareware fee is $10. The author will accept payment in any currency, like Swedish crowns, new PD programs, clean disks and so on.

PathAss is available in CompuServe's AmigaTech Library 14.

Terminal programs used to be a popular item for shareware authors to write. I am not sure why we are not seeing as many new ones now as in the past. Maybe it is because attention was turned to creating the ultimate directory utility or the ultimate file requester. Or maybe, the competition was getting very stiff, especially from Keith Young's Access! program.

There has been a lot of talk on the networks recently about the state of Amiga terminal software. Many contend that none are up to level of the communications software available for other machines. It seems that someone with the ability to implement the asked-for features was listening.
There is now a new offering from Jack Radigan called JR-Comm. This program is still in a pre-release version. As of this writing it is labelled version 0.91. I am not sure what pre-release means here. I suppose it means that the program is not yet in a completed state (is any program ever finished?) and the author has made it available as a beta test release. The public will help pin down the elusive bugs and make suggestions for future enhancements. Even in its incomplete state, JR-Comm is more powerful and feature laden than most other freely distributable and commercial terminal programs.

Some of JR-Comm’s more subtle features are: the ability run from a workbench window or a 1, 2, 3 or 4 bit-plane custom screen, the ability to send a break signal (the length of the break signal is user definable), real time and session time clocks, force a modem hang-up, user sizeable review buffer, and clear screen.

JR-Comm offers a wide variety of file transfer protocols: ASCII, CompuServe’s B+ protocol, XMODEM, XMOMED, XMODEM-1k, YMODEM, YMODEM-5, and ZMODEM. Both CompuServe’s B+ protocol and ZMODEM allow for the resumption of a transfer that had been partially completed due to loss of carrier. For use on packet switched networks, you can relax the timing. This increases the timeout period before a NAK is sent. This is necessary for non-windowed transfer protocols such as XMODEM.

When using XMODEM, XMODEM-1k, or XMODEM, auto-chop can be set. These protocols pad data to the last block of a file transfer. Auto-chop removes this padding from binary files. You also have the option of saving to disk what was received in a failed file transfer. This is necessary for use with the transfer resumption capabilities of CompuServe’s B+ protocol and ZMODEM. It is a nice feature to use with other protocols as well.

Should the file that failed the transfer be in a librared form (ARC, ZOO, etc.) you could more than likely salvage some of the files it contained. The selection of protocols is by far the largest available in a non-commercial program.

JR-Comm is an almost 100% complete IBM ANSI emulator. The default colours are set to the normal IBM colour palette of black, red, green, yellow, blue, magenta, cyan, and white. A 16 colour screen is available for use so that the high-intensity attribute is available for each of the eight IBM colours. The blinking character attribute of the IBM display is rendered as a bold character in this emulation.

An option for selecting the text and background colours of the display is given so there is no need to change the colour palette itself. An ibmfont file is included. The IBM character set can be used to complete the IBM ANSI emulation.

The phonebook in JR-Comm is one of the program’s strongest features. Phonebooks in other programs will bring in a new set of macros, change colours, and change serial parameters. Many times this is not enough. When a BBS or network is called, you might find the need to change some other aspect of the program’s settings that are not affected by the phonebook. JR-Comm allows the complete reconfiguration of the system. Every user configurable item JR-Comm offers is attached to the phonebook entries.

The phonebook offers a unique password feature for each entry. This is a nice feature for those who call a large number of BBS’s. It allows a unique password to be generated for each entry in your phone directory. Why would you want to do that? Most people who regularly call a large number of BBS’s will use the same password on all of them, just because it is easier to remember one or two passwords. On some BBS systems the passwords are case sensitive. Even if only one or two different passwords are used for the systems called, you may not remember whether it was upper-case, lower-case, or mixed case on that system.

Another good reason for allowing JR-Comm to generate the passwords for you is security. Should your password become known your security is breached on the systems you frequent. In the phonebook, there is a string gadget labelled Master password:. You supply the master password and the program generated passwords are derived from it. You can specify the length of the generated password and whether it is alpha, numeric, or both.

JR-Comm is capable of multi-dialling any number of directory entries. The number of retries per selected entry, the length of time to wait for carrier detection, and the delay period for restarting the dialling loop are all user-selectable.

The dialler has a certain amount of intelligence built into it. Should the program receive three NO DIALTONE responses from the modem the dialler will halt. The entry will be removed from your selected dial list if the modem returns a VOICE response. Should three NO CARRIER responses be received for a given entry that entry will be removed from the list. When a connection is made the entry is removed from the selected list. This will allow the entry to be skipped when the dialer becomes active again at the termination of the current call. When a connection is made, all parameters for that phonebook entry will be in effect.

For use in the conference mode of commercial services JR-Comm offers a chat mode. This is an area of the screen to type into that will not be overwritten by text coming into the main portion of the screen. It is set up so that you can type and not worry about exceeding the maximum line length the service allows. The line will automatically be sent when you reach the length limit of the chat window. The chat mode offers a history buffer. By using the up and down arrow keys you can re-display text you have previously sent. When the previously sent line is displayed all of the normal line editing features are available. The text is then sent to the modem by entering a carriage return.
As with most of the more recent terminal programs, JR-Comm offers F-key macros. You can define up to 20 of them for each phonebook entry. You may load in another set of macro definitions at any time. You cannot, at this time, create a macro that will load in another macro set.

One of the features many people have been looking for in a terminal program is the addition of a log file. This is a text file which records the activity of the program. JR-Comm creates a log file. Each time the program makes a connection an entry is made into the log. The program will log the JR-Comm session start and end times, connection via the dialer (rather than manually typing the number to dial), modem hang-up command, carrier drop, and successful downloads with the file size, total error count and character per second rate for the file transfer.

JR-Comm has a great deal of potential, and, as is, it is more powerful than most currently available communications programs. It is not without its flaws however. Some are bugs, some are design flaws.

Currently, JR-Comm does not support interface mode. Nor does it support non-standard screen sizes. This is a real annoyance for people using PAL systems or have increased their standard screen size with a utility such as MoreRows.

The chat window does not handle control characters in a way that is useful for me. Because the use of control characters is fully documented I would have to consider this a design flaw. To send a control character from the chat window you must type a \^ (caret), then the letter. For a control-P you type ^P and hit return. This is very inconvenient. If there is large amount of text scrolling across the screen the normal procedure would be to hit control-S to stop the scroll and control-Q to continue. With JR-Comm's approach, you must find the ^ (caret) key (I'm a touch typist but still must look at the keyboard to find the symbols), type the letter, and hit return. By the time I do this what I wanted to see has scrolled off the screen. Another problem is that not only is the control-P sent but the carriage return also. In many circumstances I do not want to send a carriage return along with a control character.

A nasty side effect of this approach is that the ^ (caret) alone cannot be sent from the chat window. While this is not a major problem is does cut down on the characters available for use by all of the ASCII artists out there. I much prefer the approach that Access! takes. All control characters are sent from the chat window. No prefixing with another key is necessary. Just hold the control key down and type the character.

In this version of the program there is a serious bug in the serial routines. Any time text is output to the screen all other Amiga functions cease to operate. The mouse freezes, left-Amiga-M fails to function, keystrokes are inhibited - all functionality other than text output is stopped. The events are buffered but it may take the program 10-15 seconds to act upon them. Mr. Radigan is aware of this problem and will hopefully have it solved in his next release.

An IBM font is included to fully utilise the IBM ANSI emulation. I would have much preferred that this font be internal to the program than having to be placed into FONTS:. Many programs are requiring the use of special fonts. I sometimes think that software developers forget that not all users have hard disks, or even 2 floppy disk drives. They can run out of room to keep a forever expanding FONTS:

There is an excellent utility written by Gregg Tavares of Cinemaware that will take a font and generate C code for inclusion into your source code. While this will nominally expand the size of the executable file it solves many problems. The user no longer has to find room for the needed font(s). Mistakes in the user creation of a font directory and the copying of files to FONTS: are eliminated.

It goes almost without saying that I would very much like to see full AREXX support in a future version of the program. User selectable fonts is another high priority item I would like to see supported.

JR-Comm has the potential to be one of the top two or three communications programs, freely distributable or commercial. Mr. Radigan seems to be very committed to improving and expanding its capability. In its present form (version 0.91) I would not recommend it for exclusive use, but do use it when you need the protocols it offers. Definitely try future versions as bugs are nailed down and even more functionality is included.

JR-Comm 0.91 is a copyrighted work. It is freely distributable with no shareware fee asked. When JR-Comm reaches a 1.0 release status it will more than likely become shareware or commercial.

JR-Comm is available in CompuServe's AmigaTech Library 4 as JRCOMM.ARC.

The author, Jack Radigan, may be contacted by sending CompuServe Email to 76545,201 or by calling his BBS, the Atlantic County Amiga BBS, (609)625-2453.

Stop Press

In TransAmi volume 1, issue 5, I mentioned that the CAD software by SoftCircuits would be entering the shareware arena.

While that information was correct at the time it is no longer so. A new commercial product, Board Master, based upon the SoftCircuits products will be entering the marketplace shortly. It will be priced at under $100.
Assembly Language

Part 2 - command line arguments and a twist in ECHO's tail

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.

Last time, we covered a lot of ground - this time I'd like to touch a few highlights, and add a little perspective. In part one, we had to gallop hard to get to the point where a working program could be written.

Exploring is fun, but it's time to read through this list, ensuring that our information base is well in place.

1 - The 68000 can reach memory either one byte at a time (any address) or two bytes (a word) at a time (low byte at an even address, high byte at the next higher odd address). An instruction can ask to handle four bytes (a long word), in this case there will be two dips into memory.

Thus, I could ask the processor to give me the long word contents of address 4, and it would fetch the contents of bytes 4, 5, 6 and 7, placing the result into the register you specified. If you tried to do a long word (or word) operation on address 5, however, the computer would refuse, and you'd meet the guru.

2 - Data registers may be used in either of three ways: as the full long word, all four bytes; as a word, the two lower bytes; or as a byte, the lowest byte of the register. The part of the register you don't use will not be affected.

Address register operations change the whole long word. This is true even if you specify .w (word) mode; the two bytes you move will expand to fill the whole address register. And you can't use .b (byte) mode with an address register.

3 - Address register 7 is the stack; don't change it without reason. Address register 6 is used in the Amiga for library calls; keep it fairly free for this use. Address registers 0 to 5 inclusive, and all data registers are free for you to use. You'll tend to use the higher numbered registers; that's because subroutine calls can lose the information in lower numbered registers, as the next item notes.

4 - System subroutines (library calls) are careful not to change the contents of address and data registers 2 to 7; values you have there are safe. But registers A0, A1, D0 and D1 are likely to be changed by these subroutines; you'll lose any information you were holding in these.

5 - The 68000 can reach memory either one byte at a time (any address) or two bytes (a word) at a time (low byte at an even address, high byte at the next higher odd address). An instruction can ask to handle four bytes (a long word), in this case there will be two dips into memory.

Thus, I could ask the processor to give me the long word contents of address 4, and it would fetch the contents of bytes 4, 5, 6 and 7, placing the result into the register you specified. If you tried to do a long word (or word) operation on address 5, however, the computer would refuse, and you'd meet the guru.

6 - We have met a few 68000 instructions. For moving data, we have move, which will move information (byte, word, or long word). lea (load effective address) moves an address (as opposed to moving data) into an address register.

These instructions have two operands, a from and a to, or more formally, source and destination. move usually specifies a data length, such as .L for long word, .w for word, or .b for byte. Thus, move .L 4, A6 moves the four bytes from addresses 4, 5, 6, and 7 into address register 6. lea always moves a full address, which is a long word.

BSR (Branch subroutine) calls a subroutine that is sure to be close by (within about 32,000 bytes). If the subroutine were really close, within about 120 bytes, we may save a little memory by specifying bssr (branch subroutine, short). In contrast, jsr (jump subroutine) is able to call a subroutine anywhere in memory, perhaps one that is too far away to be reached by bsr.

BSR and JSR have one operand: the location of the subroutine you wish to call.

Rts returns from a subroutine to the program that called it. The entire program that you write is a subroutine, called by the system. When it's finished, it should return control to the system with rts. Rts (return from subroutine) has no operands; it's complete in itself.

7 - We have met a few address modes. Let's summarise the most visible ones.
Register direct means the contents of the register itself. Thus, \texttt{MOVE.L A0, D0} means ‘move the contents of A0 into register D0’.

Address register indirect means that the address register contains the address of the data. Parentheses are used to indicate an indirect address; only address registers may be used this way. Thus, \texttt{MOVE.L (A1), D0} means ‘A1 contains an address; move the contents of that address into register D0’. In this example, the address in A1 must be an even number or else ...

Address register indirect, with displacement asks for an indirect address again, but does a little extra arithmetic, offsetting (or ‘displacing’) the address by a fixed amount. Thus, \texttt{JSR ~$228 (A6)} means ‘get the address from A6, subtract hex 228, and do a subroutine call to whatever address results’.

Absolute specifies a memory address in the Amiga. We rarely use fixed addresses, except for address 4, which contains a pointer to our Exec library. Even the input/output and system chips should not normally be referenced directly by the programmer; instead, a library call will get this data for you.

\texttt{MOVE.L 4, A6} calls for the contents of address 4 (and 5, 6, and 7) to be moved into register A6. Note that this is different from \texttt{LEA 4, A6}, which would move 4, the address, into A6.

Amazingly, we may use absolute addresses to refer to something within our own program. At first, this is puzzling, since we don’t know where our program will be when it’s loaded into memory. However, everything works; the assembler and Amiga’s loader work together to adjust the address so that it is correct within your program when it runs.

Thus, if we have a byte of information stored at address \texttt{VALUE} within our program, we might code \texttt{MOVE.B VALUE, D5} to transfer the contents of \texttt{VALUE} into the low byte of register D5. Whenever we can, however, we avoid this kind of absolute address in favour of PC addressing, to be described next.

Program Counter Indirect With Displacement is often called relative addressing. It tells the processor to look ahead or back from its current working point (up to about 32,000 locations either way) to find the location of interest. If you are referring to data within the current block of your own program, you will often use this addressing mode.

You can’t always use this addressing mode, but when you can it saves memory and computer overhead. Since the offset can be described in two bytes, and a full address would take a long word of four bytes, you save space and time. And this kind of address is valid no matter where your program is loaded. In contrast, an absolute address referring to a location within your program would need to be adjusted according to where your program ended up.

Our previous example of \texttt{MOVE.B VALUE, D5} could be usefully changed to \texttt{MOVE.B VALUE (PC), D5}. The code would be shorter, and would not need relocating as it loads.

The (PC) mode, however, may be used only as part of a source operand, that is, only as the first address. If our example wanted to move the other way, we’d have to write \texttt{MOVE.B D5, VALUE} since (PC) would not be available.

Immediate addressing supplies an actual value to be used. It is signalled by the use of the \# character (sometimes called the octothorpe or pounds symbol). Thus, \texttt{MOVE.L #0, A4} would put the actual value zero into register A4.

As with (PC) type addressing, immediate addressing may be used only as part of the first operand. So if I want to compare the contents of D4 with a value five, I must write \texttt{CMP.L #5, D4} ... not the other way around.

There are other addressing modes, but we’ll introduce them as they arrive.

8 - There’s a need for startup code, a set of instructions that do necessary overhead work. This includes such things as opening system libraries before the main job starts, and closing them after it is finished. Perhaps it would be more accurate to call it startup-and-closedown code.

The startup code we wrote last time assumed three things. First, that the program would be called only from the CLI. That saved us quite a complex series of operations. Among other things, we could be confident that there would be a window to which to which we could send output.

Secondly, it assumed that the only library we would need would be DOS. For beginners, that’s handy; DOS is the simplest output path to screen, disk, printers or whatever. DOS is not much good for drawing pictures or playing music; but there will be time for that later.

Our third assumption one which we will change today is that the CLI command consists of the program name only. That is, the program would be invoked by typing TEST. If we should choose to type TEST THIS PROGRAM, the remainder of the line, THIS PROGRAM, would be ignored. We’ll start to look at the question of reading the ‘command tail’, as it’s called, in this session.

9 - An observation: in writing a program that is completely self-contained, we’ve done something that’s quite unusual in Amiga-land. Most programs call in extra data definitions from \texttt{include} files as they start. Many programs use \texttt{macros} to generate canned code that doesn’t need to be repeated. It is common for Amiga programs to call in pre-written startup code. We’ll do that later, after we have written more complete startup sections. Many programmers write and assemble complete sections of code; later, the linker will combine this code with other segments to make a complete operating program.

Finally, it’s almost universal to ask the linker to look up the library addresses for you; it’s too much work to remember that $228 is the \textit{OpenLibrary} entry.
We'll open up this area only slightly at the moment. There are different rules that apply to different assembler/linker systems. We'll control as much as we can, so that there's minimal dependence on any special system features.

Next project: reverse ECHO

We know that CLI command ECHO repeats what it finds in the command tail; at least, that's its main job. Our project will be to write an ECHO command that repeats the information backwards. Not very useful, but we'll learn a few things on the way. We will call this program OHCE... ECHO spelt backwards.

Open up a new file called OHCEASM on your favourite editor, and let's get to work. A few comment lines, and then we'll do our system library definitions. They will be the same as in the earlier program; but this time, we'll let the system figure out their values.

```
; Reverse ECHO program
; Startup sequence, CLI only
; Definitions for Exec Library
XREF _LV00penLibrary
XREF _LV0CloseLibrary
; LVO means Library Vector Offset

XREF tells the assembler: 'Leave a blank any time you see these labels, together with a note to the linker that this value needs to be filled in'. The assembler will then go ahead with the job; even though it doesn't know the value of, say, _LV0OpenLibrary, it won't signal an error.

The linker, in turn, will read the object file and understand that it will need to find these labels in some other file: another object file or a library file. When it finds the labels, it will fill in the correct values. Result: we end up with a complete executable program.

You can't use XREF with the AssemPro assembler, which doesn't do linking. Instead, substitute the command:

```
INCLUDE INCLUDES/EXEC.OFFSETS
```

This will automatically insert a set of EQU definitions for _LV0OpenLibrary and _LV0CloseLibrary, plus many others we won't need for this program. We'll do this again when we come to the DOS library definitions.

Catching the tail

Our previous startup-sequence began by opening the DOS library. We have something new to do before tackling that. Information on the command tail is given to us by the Amiga loader: AO contains the address of this text string, and DO contains its length.

Registers AO and DO will have their values disturbed when we call a system subroutine. So we'd better park their values somewhere safe. Any register numbered 2 or higher will do the job, so we'll code:

```
MOVE.L A0,A5
MOVE.L D0,D5
```

We could have picked any other convenient registers for these values, keeping in mind that A6 and A7 have other functions to perform. Now we may safely open the DOS library.

```
MOVE.L $4,A6 ; Exec base
LEA DosName(PC),A1 ; Dos name string pointer
MOVE $0,D0 ; any version
JSR _LV0OpenLibrary(A6)
MOVE.L D0,A6 ; Dos base pointer
BRQ Exit ; Zero, quit
BSR S Main ; Do the job
MOVE.L A6,A1 ; Dos base pointer
MOVE.L $4,A6 ; Exec base pointer
JSR _LV0CloseLibrary(A6)
```

```
Exit:
RTS
```

This completes our startup code. It's about the same as we have used previously, except for the XREF and the saving of our command tail registers.

On to the main job. As before, we'll make the library definitions into XREF statements, and then call to get the output stream file handle.

```
Main:
; Main Job
; Definitions for Dos library
XREF _LV0Output
XREF _LV0Write
```

Again, you can't use XREF with AssemPro. Substitute:

```
INCLUDE INCLUDES/DOS.OFFSETS
```

This will insert EQU definitions for _LV0Output and _LV0Write, plus many others. These OFFSET files are in text form; you can read them if you like.

```
JSR _LV0Output(A6) ; Get output handle
MOVE.L D0,D4 ; Handle to D4
```

We'll need to use our handle more than once ... in D4 it will be safe from subroutine erasure.

Remember our command tail? Our job is to print it out backwards. We have the location of these characters in address register 5, and the number of characters in data register 5. The characters are bytes, of course.
We would use Write to print our character, of course. There's no problem printing the characters one at a time, and that's what we'll do, using a loop. More sophisticated programs would copy the reversed message into a work area of memory, and write the whole thing at one shot.

By the way, the command tail string is not terminated by a zero byte. You'll usually find a newline (linefeed) character there. If you're testing for end-of-tail, it's usually safer to look for any non-printing character (any value less than 32 decimal). But we don't need to do that, since we know the length of the string: it's stored in D5.

I think we can safely assume that the string is less that 65,535 characters long. That means that we won't need to handle D5 as a long word; a word operation will get the complete value and save us a little (negligible) amount of run time.

The SUB (subtract) instruction will subtract 1 from D5. In this case, a small value is involved, permitting us to use the SUBQ (subtract quick) command. If the result goes below zero we will Branch Minus (BMI) to the loop exit. Otherwise, we have something to print:

```
LEA 0(A5, D5.W), A0 ; Address of Character
```

Aha! A new addressing model: this one is Address register indirect with index and 8-bit displacement. That's a mouthful, but you may be able to see what it does by just looking at it. The address in A5 is added to the counter value (word size) in D5; then zero is added to the whole shebang. On the first time round the loop, this gives the address of the last character of the command tail. As we go round the loop, decreasing D5, we'll back to previous characters. Of course, Write needs that address placed into register D2, so we take an extra step.

```
MOVE.L A0, D2 ; .. to D2
MOVEQ $1, D3 ; Length of Character
MOVE.L D4, D1 ; Handle
JSR _LVOWrite(A6) ; Send Character
BRA.S Loop
```

To be neat, we must print a newline before the program terminates. Can we assume that register D3 still contains value 1?

```
Finish:
LEA NewLine, A0
MOVE.L A0, D2
MOVE.L D4, D1
JSR _LVOWrite(A6)
RTS
```

The whole thing

Here's the whole program in one piece. Keep in mind that labels (sometimes called symbols) must be hard against the left hand column for most assemblers to treat them properly. Some assemblers also want comment lines to start at the left margin.

```
; Reverse ECHO program
; Startup sequence, CLI only
; Definitions for Exec Library

XREF _LVOpenLibrary
XREF _LVOCloseLibrary

; LVO means Library Vector Offset

MOVE.L A0, A5
MOVE.L D0, D5
MOVE.L $4, A6 ; Exec base
LEA DosName(PC), A1 ; Dos name string pointer
MOVRO $0, D0 ; any version
JSR _LVOpenLibrary(A6)
MOVE.L 00, A6 ; Dos base pointer
BEO.S Exit ; Zero, quit
BRR.S Main ; Do the job
MOVE.L A6, A1 ; Dos base pointer
MOVE.L $4, A6 ; Exec base pointer
JSR _LVOCloseLibrary(A6)

Exit:
RTS

Main:
; Main Job
; Definitions for Dos library

XREF _LVOutput
XREF _LVOWrite

JSR _LVOutput(A6) ; Get output handle
MOVE.L D0, D4 ; Handle safe in D4

Loop:
SUBO.W $1, D5
BMI.S Finish

LEA D(A5, D5.W), A0 ; Address of Character
MOVE.L A0, D2 ; .. to D2
MOVEQ $1, D3 ; Length of Character
MOVE.L D4, D1 ; Handle
JSR _LVOWrite(A6) ; Send character
BRA.S Loop

Finish:
LEA NewLine, A0 ; Address of NewLine char
MOVE.L A0, D2 ; .. to D2
MOVE.L D4, D1 ; Handle
JSR _LVOWrite(A6)
RTS

DosName dc.b 'dos.library', 0
NewLine dc.b $a
end
```
Assembling

Type it all in and save it, using the name OHCE.ASM. Again, be careful about upper and lower case names. No need for a printout yet ... the assembler will usually do that for you. Keep in mind that if you're using AssemPro, you must change the XREF lines in favor of INCLUDE statements, and you will skip the BLINK step described below.

Now start the assembly. On some packages, that means selecting Assemble from a screen menu. Other assemblers will call for you to type a CLI command. You might type:

(MetaComCo): ASSEM OHCE.ASM -o OHCE.OBJ
(Charlie Gibbs): A68K OHCE.ASM OHCE.O

Note that we specify the source (.ASM) and object (.O or .OBJ) file names; I have not supplied a full path name, since you might choose to put these files in RAM:, DP0: or even hard disk. To get a listing file, you would expand the commands slightly:

(Mcc): ASSEM OHCE.ASM -o OHCE.OBJ -1 OHCE.LST
(CG): A68K OHCE.ASM OHCE.O OHCE.LST

The file names above are arbitrary, but typical. Keep in mind that CAPE will produce an object file named OHCE.OBJ only if you ask for it; otherwise you'll get AsmObjTemp. Small.lib works better (with most versions of BLink) if you leave out the LIB (library) keyword.

Running the result

A command in CLI such as OHCE DING DONG will produce the satisfactory response of GNDD GNID. We've 'twisted the tail'.

Here are some things for you to try, or to think over. If we redirect the output, as in

OHCE >RAM:XX DING DONG

will the redirection characters be part of the command tail? If so, you'll see them in the output file. Will extra spaces at the beginning or end of the command and its tail be preserved or thrown away?

You might like to verify that RUN OHCE DING DONG works as expected. You could even try executing a script file containing a command OHCS1 to ensure that everything works that way.

Lastly: if we type OHCE DING DONG, how many characters will be in the command tail? Will the newline character at the end of the line be included in the count (in register DO), or not?

Variations and documents

You may have an assembler other than the ones named above, but you'll find that the principles are the same. In case you need to define your own _LVO values, the following diagrams give you the entry point offsets in decimal, hexadecimal, and two's complement hex. Your debugger or disassembler might use any one of these numbering systems.

Don't get swept away buying every Amiga reference text you can find ... yet. It would cost a lot of money, and take up a lot of shelf space. Wait until you decide upon the areas you will need to investigate in detail. In the meantime, there's a handy form of quick library documentation that's free to all Amiga owners: the FD files.

The FD files are on your Extras disk. They are in text form; you can list them with a CLI command such as TYPE. They will give you a hint as to the fields you will need to set up, and the registers involved.

BLink (a successor to the original ALink), by John Toobes of The Software Distillery, is freely distributable and almost universally used for both assembly language and C programming.
Some entry points in dos.library

- $FF22 -$DE -222 Execute
- $FF7C -$84 -132 IoErr
- $FF82 -$7E -126 CurrentDir
- $FFA6 -$5A -90 UnLock
- $FFAC -$54 -84 Lock
- $FFC4 -$3C -60 Output
- $FFCA -$36 -54 input
- $FFD0 -$30 -48 Write
- $FFD6 -$2A -42 Read
- $FFDC -$24 -36 Close
- $FFE2 -$1E -30 Open

Some entry points in exec.library

- $FFD8 -$228 -552 OpenLibrary
- $FE62 -$19E -414 CloseLibrary
- $FE80 -$19E -384 WaitPort
- $FE86 -$17A -378 ReplyMsg
- $FE8C -$174 -372 GetMsg
- $FEDA -$126 -294 FindTask
- $FF2E -$D2 -210 FreeMem
- $FF3A -$C6 -198 AllocMem
- $FF76 -$8A -138 Permit
- $FF7C -$8C -132 Forbid
BreakPoint Part Five

Debugging with DB and MetaScope

by Victor A. Wagner
Copyright ©1989 Victor A. Wagner, Jr.

Vic Wagner started with computers in 1965 while in the US Air Force in a group studying digital flight simulation. Since becoming a civilian in 1966, he has been working mostly with minicomputer manufacturers on real-time software systems. Vic spends his days answering the Computer Automation Inc. technical support line and maintaining real-time software systems. In his spare time, he talks to Taarna (his Amiga 1000), writes programs and spends time on CompuServe's Amiga forums (76046,3004) where he is currently teaching a C class.

In this last installment we're going to take a look at the remaining two readily available debuggers in the Amiga marketplace. These two are old-timers in the Amiga debugging arena and are about as different in user interface as two programs could possibly be. DB, from Manx Software Systems, uses primarily a text interface; MetaScope, from Metadigm, is completely intui-
tionised (menus, gadgets, point and click). Both are broadly equivalent in abilities.

Before we go on, I should point out that I have been using MetaScope since before it was publicly released, and that I have some financial and emotional interest in its success. I've known the author for over ten years and, throughout MetaScope's design and implementation we spent many a lunch hour scoffing pizza (the only true food of programmers) and discussing features and options. I have done extensive testing of MetaScope and am the author of the demonstration text on the current release. That MetaScope follows (to some extent) my prejudices in debugging techniques is therefore not surprising, although some of its features drive me to distraction. I will endeavour to give as unbiased a view as possible.

Okay, let's look at some of the physical aspects of the debuggers before we enmesh ourselves in the features of the programs themselves.

MetaScope: Version 1.23

The versions on my system are:

<table>
<thead>
<tr>
<th>MetaScope</th>
<th>db</th>
</tr>
</thead>
<tbody>
<tr>
<td>81336</td>
<td>60360</td>
</tr>
</tbody>
</table>

This version of MetaScope is a beta version which has source compatibility with the Lattice compiler, and by the time you read this should be working with the Manx compiler as well.

As you can see, MetaScope is about 35 per cent larger on the disk. In memory they look like:

<table>
<thead>
<tr>
<th>MetaScope: size</th>
<th>DB: size</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>Hex</td>
</tr>
<tr>
<td>0267D8</td>
<td>04</td>
</tr>
<tr>
<td>87A5D0</td>
<td>3398</td>
</tr>
<tr>
<td>8CDAC8</td>
<td>103CD</td>
</tr>
<tr>
<td>8921A0</td>
<td>32D0</td>
</tr>
<tr>
<td>1382C</td>
<td>79916</td>
</tr>
<tr>
<td>89B8D0</td>
<td>22C8</td>
</tr>
<tr>
<td>8BA598</td>
<td>10A4</td>
</tr>
</tbody>
</table>

MetaScope is about 20 per cent larger when in memory. Remember that the Amiga does scatter loading and therefore the addresses have little meaning.

The other physical differences are in the manuals. DB's manual is 42 pages, 3-hole punched and included in the binder with the Manx compiler. MetaScope's manual is a 35 page centre-stapled booklet 5.5" x 8.5". Neither manual has an index, but both have tables of contents and are written in the style of computer program manuals; each is more like a dictionary rather than a novel.

Considering the difficulty I am having writing about debugging in general, I have a great deal of empathy with the authors of these manuals. There is a lot of information to impart to the reader, and most of the information is rather diverse and disconnected. All debuggers seem to be collections of things which programmers have found useful over the years. This should not be taken as negative criticism for the manuals, but rather as an explanation as to the relative disconnectedness of the manuals.
Even casual perusal of the manuals will point out the vast difference in approach to operator interface. DB is strictly textual like most of the debuggers which preceded it. DB opens with a single window and its log-on message and a prompt. DB gains control of a task by using the al or aL command. This puts some hooks into AmigaDos and grabs onto the next program to be executed, either from the CLI or the workbench. After DB gains control, it prints the registers and awaits further commands.

MetaScope, on the other hand, opens up what it calls a status window and then just sits there. There is not even any place to type. MetaScope (MS from now on), however, is a very Amiga-gatized program; there are menus on it's window, one of which allows you to load a program. MetaScope also allows you to include the command line for the program you're debugging on the MetaScope invocation line as in:

```
>MetaScope df0:proja/aargh -a -b -c
```

The above example was taken from the MS manual. DB, on the other hand, has a default file which it will read upon invocation, and if it contains the single command al (Amiga, load with symbols) then he following does the same thing:

```
>db
>df0:proja/aargh -a -b -c
```

Rather than give you a blow by blow of what each debugger can do, I'm going to tell you the things that each can do that the other cannot.

First, I should point out that both debuggers have rather complete expression evaluators. The MS manual takes 2-1/2 pages to detail its expression capabilities (resembles the C language a lot). DB takes about three pages. I wouldn't bet a nickel on either of them being more powerful than the other.

Okay, on to the specifics. I'll go through the DB commands in roughly alphabetical order and comment on those things missing from MS.

DB has many Amiga specific command (they all start with a) for which there are no counterparts in MS. Many (mayhap all) of the exec lists can be displayed; devices, interrupts, libraries, ports, resources, and tasks.

The al command is particularly useful, especially if you are debugging multiple tasks. When DB is started it opens one window for dialogue. With this command you open another window and can then debug two tasks at the same time. Of course, you use the al command to capture the next task to start.

DB has breakpoints which watch the value of memory locations, and will then stop execution if the memory either becomes equal to (or NOT equal to) a specified value. The length of time it takes DB to recognize that the memory location has changed depends on the mode DB is in at the time. If you are single-stepping the detection is immediate; if not, these memory change breakpoints will be examined at each function entry and exit. DB also has the ability to hash or checksum a region of memory and to check this periodically as above. Explicit warning against doing this is to the first 256 bytes of memory is given since this would slow down the ability to single step. There is a special command to watch the first 256 bytes of memory.

DB has a breakpoint feature that I'm sure every programmer has wanted at some point. You can set the address of a function which DB will call every time it regains control. If the function returns zero, then it is as if nothing had happened. But if it returns a non-zero value, the program is stopped, and a message is displayed showing the value returned.

DB allows two temporary breakpoints to be set and allows temporary breakpoints either with or without the breakpoints in the breaklist being activated.

By allowing a command list to be executed upon reaching a breakpoint, selected memory locations can be displayed when the program is suspended.

DB devotes almost six pages to the prin command. This allows you to format data in almost any conceivable fashion. The formats seem to be a superset of printf(), and about as cryptic.

There are commands for allocating (perhaps for patches), comparing, filling, and moving memory and up to 26 macros can be saved (one per letter on the keyboard). This should save a fair amount of time as reading keys from memory is much faster than I can type.

10 can be redirected so that commands can be input from a file, and the output which goes to the window can be directed to a file. Especially useful is the ability to log commands only to a file, so that you can read them in again later, and retrace the steps being done now.

The first thing you notice when you start playing with MS is that there appears to be no limit to the number of output windows which may be opened. Although broadly true (since each window takes some memory, there is a limit based on available memory), the usefulness of having multiple break, hunk, status, or symbol windows is of limited usefulness.

Having multiple memory windows, on the other hand, alters one's method of debugging. If you've been used to supplying commands to display some memory locations each time a breakpoint is reached, that is unnecessary in MS (it's impossible as well). You simply open a window on the requisite memory, and MS will display it. Before you say 'Wait a minute, I use my commands to display things based on register (or variable) contents' I should point out that MS allows its full expression capability to determine the start address of a window. This expression can also be either static (determined at the time it is entered) or dynamic (determined each time the window is to be displayed. Particularly useful expressions are:
pc - the current program counter - useful for seeing where the program currently is and sp - the current stack pointer - for looking at all those arguments to functions, and whatnot.

Of course, you’d want these expressions to be ‘dynamic’ so that you can see the memory as the registers change. The use of the word ‘dynamic’ might be slightly misleading since the expressions which contain registers are evaluated only when MS has control of the program. While this allows you to look at memory referenced by registers, it doesn’t waste a lot of time re-displaying different regions of memory when your program is actually running.

‘But certainly, MS doesn’t display changing memory while my program is running, and MS isn’t in control’, I hear you ask. Well, that may be the biggest difference between MS and the other debuggers we’ve looked at.

MS updates its memory displays around ten times a second independent of whether the program is running or not. This means that you can look at such things as the mouse x & y variables in a Window structure and watch them change as you move the mouse around.

MS has scroll bars and sizing gadgets on all of its windows so that you can change the amount displayed as well as scroll around and you can select addresses to be examined or used as breakpoints just by pointing and clicking on them.

Conditions can be placed on permanent breakpoints. These are expressions which are evaluated whenever the associated breakpoint is reached. If the expression is true (non-zero), then the counter associated with the breakpoint is incremented and, if the required count has been reached, the break actually happens. This means that MS can produce breakpoints like: ‘break here the third time you get here and the D0 register is equal to the contents of the beginning of the input buffer’.

MS has a freeze feature for both status windows and memory windows which keeps the contents of the window constant for later comparison with the current state. It can then restore the machine to the state contained in a frozen window. This means that you can save the state of the machine (by freezing the appropriate windows) then restore it later if you want to go back and try it again.

You can have MS automatically record all of its actions in a file (or to the printer). You can also dump a window to the log.

With MS, you can title your many windows with such imaginative names as ‘Where we are now’ (for a window with dynamic base address of pc), or ‘Stack’ (for, of course, dynamic base of sp), and this can be a lifesaver. Of course when MS writes such a window to the log, the title is included.

MS allows you to change a value in memory (or a register) just by double clicking on it. A requester appears and you type in a new value.

MS allows you to change an instruction in memory by typing in assembly format. If you have a memory window open which is displaying code format rather than data format, Instructions may be double clicked and altered in assembly form.

The next couple of features are available only in the newest (and as yet unreleased) update of MetaScope. The author assures me that this update will be available by the time you read this.

MS can save a configuration of: permanent breakpoints, window locations, sizes, base address expressions, and status (frozen or not). Thus you can set up your debugging session as you like, then save it so you don’t have to retype it all in each time you start a new session.

MS allows you to intermix source display with its disassembly in memory windows so now you can see what your compiler is doing. Clicking on a source line is equivalent to clicking on the first address of generated code for that line.

Well, that about wraps up the differences between the debuggers. Whether you like windows and point and click or mostly a text environment is personal preference. I think having memory continuously displayed is an extremely handy feature, but I’m sure there are others who prefer ‘command lists’ to be executed at each breakpoint.

In conclusion

I hope you’ve enjoyed reading this series of articles. I’ve had fun writing them. Perhaps after MetaScope is released with full source capability, and Avant-Garde gets it’s M2 debugger ready, we can again take a look at ideas in debugging, and at the latest features - I presume that Lattice and Manx will have updated CPR and SDB by then.

I’ve tried to be not too dogmatic in this series of articles. My experience with many systems, debuggers, and programmers is that each person has to pick a style with which he or she is comfortable. Debugging is still an art in many senses of the word, and if you’re comfortable with the technique, then it will work for you. If you’re not comfortable with the technique, then it will just get in the way.

I would like to leave you with one thing that helped my debugging more than any other.

Have some idea of what you expect before you run the test.

* He sprang to his keyboard
  One last break to put in
  But he started to vanish
  He was looking quite thin.

* And I heard him exclaim:
  e’er he faded from sight
  ‘Happy debugging to all.....
  Give that GURU a fight!"
Multiple Gadgets

Some thoughts on making duplicate gadgets

by Peter Booth

There are many occasions where it is necessary to create a set of gadgets which are identical apart from their screen locations (and perhaps some associated text). You might, for instance, be designing a set of gadgets to be used for a gadget-oriented option-menu, or creating a set of keys for a calculator keyboard display.

In these instances, it is often a good idea to generate the set of gadgets algorithmically.

The approach brings several benefits - the size of your program code is reduced (mainly as a result of losing a lot of static structure definitions), the positions of the gadgets are easily changed, and of course you need only design one gadget, irrespective of the number that are required on the display.

Indeed, when it comes to tackling some specific problems, the algorithmic approach is sometimes actually the best.

For instance, you might want to create game-boards, or partition background displays into active sections using gadgets that have no imagery.

Or you may be writing a program that solves simultaneous equations and want the ability to enter the numeric data directly into the corresponding matrix definition. Here there may be a gadget for each element in the matrix and, if the program was to be capable of solving different numbers of unknowns, you would need to create a different number of gadgets each time.

For instance, if a user wanted to solve for four unknowns, you might set up a gadget display which looked something like this:

Despite a lot of variations on the basic theme, the problem is essentially that of efficiently generating 'arrays of gadgets'!

Problems such as these are solved once and for all by tackling the general case of \(N \times M\) gadget array generation. To create such collections of gadgets, we not only have to consider the height and depth of the gadget, but also such things as the starting co-ordinates of the array, the horizontal and vertical gaps between the gadgets themselves and so on.

The following figure should give an idea of a simple, but reasonably satisfactory, framework:

![Diagram showing a simple framework for generating gadget arrays](image)

In practice

To create a single gadget we would have to allocate some memory for a Gadget structure, set up the various parameters in the gadget fields, then add the gadget to the window’s gadget list.

To set up \(N \times M\) such gadgets, we’d just need to perform the above operations \(N \times M\) times - the only significant change being that a subscript arrangement would be needed to both set up, and access, the Gadget structures.

To generate the array we can use a doubly nested loop based on the following:
To illustrate the overall idea, I’ve sketched out the outline of some example code on the right which sets up an array of boolean gadgets.

The example doesn’t contain any error checking because it clutters what is essentially a very simple piece of code.

In practice, the main error handling would be for AllocMemQ, and this is probably best done by traversing the pointer array in the reverse order, using a count-down version of the doubly nested loop to deallocate every previously allocated memory block.

Another approach for memory allocation might be to use the AllocRemember() function for each allocation, or perhaps even estimate the total memory required, make a single call to AllocMemQ, and then partition it into the required number of individual Gadget structures (these latter two options would allow deallocation at any stage with a single function call).

What are the benefits?

Well, for a start, it’s straightforward to code and, because gadgets are positioned by calculating their co-ordinates, the approach lends itself to many possibilities.

For instance, the position of the gadgets could be based on a simple sum (as in our example), or they could be derived from more complex functions (perhaps putting gadgets around a roulette wheel by calculating the appropriate \( (x, f(x)) \) coordinate pairs).

Alternatively - predetermined co-ordinate lists could be created, and the gadget positions obtained by reading the list as the gadgets are set up.

Well, that’s the approach! In practice, it’s a concise and effective way of tackling this type of problem.

Here’s that piece of example code to get you started:

```c
struct Gadget *array[ROWS][COLUMNS];

for (i=0; i<ROWS; i++)
{
    for (j=0; j<COLUMNS; j++)
    {
        array[i][j]=struct Gadget *AllocMem(sizeof(struct Gadget),0);
        array[i][j]->NextGadget=NULL;
        array[i][j]->LeftEdge=WIDTH*j+H_GAP*j+H_OFFSET;
        array[i][j]->TopEdge=HEIGHT*i+V_GAP*i+V_OFFSET;
        array[i][j]->Width=WIDTH;
        array[i][j]->Height=HEIGHT;
        array[i][j]->Flags=GADGETCONF;
        array[i][j]->Activation=RELVERIFY;
        array[i][j]->GadgetType=BGADGET;
        array[i][j]->GadgetRender=(APTR)&border;
        array[i][j]->SelectRender=NULL;
        array[i][j]->GadgetID=1<<1+i;
        array[i][j]->UserData=NULL;
        AddCadget(global_window_p,array[i][j],0);
    }
}

RefreshGadgets(GLOBAL_window_p->FirstGadget, window_p, NULL);
```
MIDI - The Hardware

Build yourself a MIDI interface

by Steve Simpson
Copyright © 1989 Steve Simpson

Steve is a freelance consultant programmer of the Amiga and IBM PC. He is an ex-astronomer whose current interests include developing system and DOS utilities. He is currently working on an improved AmigaBridgeBoard interface, when not building Z80-based custom MIDI instruments.

The Amiga has established itself as one of the major reasonably-priced music-making machines currently available. There is a great deal of commercial music software available for it, both using the superior sound generation capabilities of the machine, and for musical instrument control.

The standard used by many instrument control software packages is the MIDI (Musical Instrument Digital Interface). This is the first of several articles in which interfacing techniques and programming for the MIDI standard are demystified.

In this article we will construct an interface which can be used with the A1000 and the A2000/A500 machines. The interface enables the Amiga to communicate with MIDI equipped synthesizers, drum machines, effects units and other peripherals. We will present a very brief introduction to the subject of MIDI, and some test programs with which to experiment.

Although MIDI interfaces can be purchased from various third-party hardware manufacturers, there is a great satisfaction in building one's own hardware devices. The MIDI interface presented here can be successfully constructed by anyone with basic electronics skills and equipment.

An etched PCB and parts kit may be made available if there is sufficient reader interest.

MIDI - a brief introduction

The MIDI standard requires that MIDI data be transmitted serially (that is, each byte is sent out one bit at a time) - an ideal job for the versatile serial hardware that already exists on the Amiga. But since the signals from the Amiga's serial connector are RS232 voltage levels, rather than the current-loop method required by the MIDI standard, we require an interface between the two.

Information is transmitted on the MIDI network using messages. Each message consists of a status byte, usually followed by one or more data bytes. A status byte is indicated by setting the most significant bit (MSB) of the byte to 1, whilst for data bytes the MSB is set to 0 - implying that for MIDI, the basic interchange level is only 7 bits.

For status bytes, the high four bits indicate the type of status message being transmitted, and the low four bits indicate for which channel the message and the following data are intended. The sixteen MIDI channels are thus represented by 0 hex to F hex (channels 1 to 16).

There are two basic types of messages in the MIDI standard: channel messages and system messages.

Channel messages are used to convey information in the 16 permitted channels available to MIDI. This information describes, for example, note-on and note-off commands, note values and velocity, pitch bend, key pressure and program change information, and controller and MIDI modes.

System messages are used to convey control information to the MIDI network, and are not associated with a particular channel, but are available to any item of MIDI equipment. MIDI system messages cover such things as MIDI time-code and start, stop and reset signals for sequencers.

In the example program midi_send.c, we send three bytes out onto the MIDI network:

byte 1: the channel message 144, (0 in the low 4 bits) indicating a note-on to MIDI channel 1

byte 2: the note-number 60 (corresponding to middle C)

byte 3: 100 indicating the note-velocity (in the range 0 to 127, and normally used by synthesizers and so on to determine the volume of the note).

We will discuss MIDI messages in greater detail at a later date. In the meantime the reader is invited to consult any of the many books covering the MIDI standard.
A MIDI network consists of a number of instruments, each equipped with a MIDI interface. The MIDI interface usually has one IN socket (for receiving MIDI data), one OUT socket (for sending MIDI data generated by the instrument) and most also have a THRU socket (which duplicates all data received on the IN socket, for passing on to other instruments).

In general, only one device (either an instrument such as a synthesiser, or a computer) can control all the instruments on the MIDI network. Therefore, the OUT from the controlling device (perhaps an Amiga with MIDI interface) would be connected to the IN of the first instrument and the THRU of this instrument would be connected to the IN of the next, and so on, as in figure 1.

---

**Figure 1:** An example MIDI network
The Amiga controls two synthesisers and a drum machine.

---

On each instrument it is possible to set a channel number and the instrument will then respond only to messages for that designated channel, ignoring others.

With some instruments, it is also possible to 'dump' data back to the Amiga, for example, to upload synthesiser voice settings.

**Operation**

The interface presented here has 3 DIN sockets; one for MIDI IN, one for MIDI OUT and one for MIDI THRU.

In the configuration shown in figure 1, the Amiga operates a sequencing program, issuing MIDI messages to the synthesisers and drum machine. Note however, that the drum machine may have to be positioned as the final MIDI device in the 'daisy-chain' since many lack a MIDI THRU socket.

Figure 2 shows another MIDI configuration in which a MIDI instrument, such as a keyboard synthesiser, is connected to the MIDI IN of the interface and may be used by the Amiga's software as a source of input data.

---

**Figure 2:** A more complex MIDI network
A MIDI instrument sends data to the Amiga, and also direct to Synth #2. The Amiga itself controls Synth #1 and a drum machine.

Using a sampled sound library and sequencing software, the Amiga itself could control other MIDI devices (such as Synth #1 and the drum machine) through the OUT connector, and also playback the sounds requested by the MIDI instrument connected to IN. It would also be possible to use a MIDI sequencing or monitoring program to monitor the input MIDI messages.

The MIDI THRU socket on the interface will allow the data from the instrument connected to the IN socket to be passed straight to another device, independently of the Amiga. In figure 2, this
means that Synth #2 is controlled exclusively by the MIDI instrument, while Synth #1 and the drum machine are controlled by the Amiga (which may itself be controlled, or at least programmed, using data from the MIDI instrument).

**Construction**

The interface is constructed on a single sided prototyping PCB. Figure 4 shows the circuit diagram and the parts list for the MIDI interface.

Connection to the Amiga's serial port is with a ribbon cable, terminated with a 25-way D-type connector. At the PCB end, an IDC connector is used to simplify matters. However, a discrete wire cable may be used if that is more convenient.

In order to cater for the fact that power is on a different pin on the A1000 and A500/A2000, a small minicon latch is used to 'switch' between the two possible sources of +12 volts.

The first stage in constructing the MIDI interface is to plan where the components are to be sited on the PCB. IC sockets should be used, especially for the opto-isolator which is a relatively costly item. Decide where the audio connectors and the sockets for the integrated circuits are to be positioned, allowing sufficient space around these sockets in which to locate the discrete components.

Construction should proceed by first mounting the IC sockets, the DIN audio sockets and the 25-way PCB-mounted IDC cable connector. Do not insert the ICs until the board has been constructed and tested.

The voltage regulator, resistors, capacitors, diodes and transistor can then be mounted. The 3-way minicon latch could be positioned close to the voltage regulator although a simple wire jumper could be used as an alternative.

Once the components have been mounted on the PCB, check for short circuits and unsoldered joints.

One side of the minicon latch is connected to the pin on the IDC connector which corresponds to pin 23 on the serial interface socket (+12V supply for the A1000), and the other side to pin 9 on the serial interface socket (the +12V supply for the A2000/A500). The centre pin on the minicon latch is connected to the IN terminal of the voltage regulator.

This means that the interface can be used with A1000 and A2000/A500 machines after moving the latch jumper connector. Figure 3 shows the pin layout at the serial interface socket on the back of the Amiga.

After wiring up the components, the 5 pin MIDI DIN audio sockets may be wired. Once again, check for any shorts made during the wiring stage. Whilst wiring up, it is easy to miss some connections, so check the construction against the circuit diagram.

**25-pin D-type 'male' connector seen from pin-side**

<table>
<thead>
<tr>
<th>PIN</th>
<th>A1000</th>
<th>A500/A2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GND</td>
<td>(chassis ground)</td>
</tr>
<tr>
<td>2</td>
<td>TXD</td>
<td>Transmit Data TXD</td>
</tr>
<tr>
<td>3</td>
<td>RXD</td>
<td>Received Data RXD</td>
</tr>
<tr>
<td>4</td>
<td>RTS</td>
<td>Request to Send RTS</td>
</tr>
<tr>
<td>5</td>
<td>CTS</td>
<td>Clear to Send CTS</td>
</tr>
<tr>
<td>6</td>
<td>DSR</td>
<td>Data Set Ready DSR</td>
</tr>
<tr>
<td>7</td>
<td>GND</td>
<td>(signal ground)</td>
</tr>
<tr>
<td>8</td>
<td>CD</td>
<td>Carrier Detect CD</td>
</tr>
<tr>
<td>9</td>
<td>+12V</td>
<td>+12V</td>
</tr>
<tr>
<td>10</td>
<td>-12V</td>
<td>-12V</td>
</tr>
<tr>
<td>11</td>
<td>AUDOUT</td>
<td>o/p left audio channel AUDOUT</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>-5V</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>AUDOUT</td>
<td>AUDOUT left audio channel AUDIN</td>
</tr>
<tr>
<td>16</td>
<td>AUDIN</td>
<td>auditory audio channel AUDIN</td>
</tr>
<tr>
<td>17</td>
<td>BB</td>
<td>EB buffered 7.5kHz clock</td>
</tr>
<tr>
<td>18</td>
<td>INT2</td>
<td>level 2 interrupt</td>
</tr>
<tr>
<td>19</td>
<td>DTR</td>
<td>Data Terminal Ready DTR</td>
</tr>
<tr>
<td>20</td>
<td>+5V</td>
<td>+5V</td>
</tr>
<tr>
<td>21</td>
<td></td>
<td>+12V</td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>MCLK</td>
<td>MCLK 2.5MHz clock</td>
</tr>
<tr>
<td>25</td>
<td>MRHS</td>
<td>MRHS buffered reset</td>
</tr>
</tbody>
</table>

**Figure 3: Amiga serial port (RS232) connections**

At this stage connect the MIDI interface to the Amiga's serial interface with the ribbon cable.

Before inserting the ICs, and with the 3-way latch jumper disconnected, check for +12V from the Amiga at the latch terminals. With the latch jumper connected for your Amiga model, check for +5V from the OUT pin of the voltage regulator. It is also a good idea to check the voltages at various
points around the circuit such as at the power pins on the IC sockets.

It should now be safe to insert the ICs into their sockets.

Testing

The `midi_send.c` and `midi_get.c` programs (in listings 2 and 3) can be used for testing your MIDI interface. `midi_send.c` sends a MIDI message to the MIDI OUT on the interface and `midi_get.c` is a simple MIDI monitoring program and will monitor the MIDI IN for incoming MIDI messages.

The `midi_routes.c` module (listing 4) contains the low-level routines for initialising the MIDI interface and communicating with it and the include file `midi.h` (listing 1) contains the constants used in these modules.

The programs are very basic and should be used for testing purposes only. They may be changed according to the reader’s own test requirements. The routines presented here `peek` and `poke` the custom chips’ registers directly [tut, tut!! - ED].

Care should be exercised when changing the routines. Because of the high communications rate employed by MIDI, routines intended for use with MIDI need to be very efficient. In a larger project, these would be written in assembler.

With a MIDI synthesiser connected to the MIDI OUT of the interface, `midi_send` will send a note to it (middle C). The program assumes that the synthesiser is set to MIDI channel 1.

Connecting the MIDI synthesiser to the MIDI IN of the interface and running `midi_get` the MIDI messages output from the device will be monitored and displayed on the screen. The program terminates on receiving 255 (or FF in hexadecimal).

A ? is output when the first byte of a received sequence is not a MIDI status byte.

Using the Interface

The MIDI interface presented here can be used as part of any Amiga MIDI sequencing and editing applications such as ‘Dr. T’s KCS’ and ‘MIDI Synergy’.

Some of the MIDI interfaces available from third-party manufacturers have more than one MIDI OUT socket. This is useful when say utilising an open-ended daisy-chain of MIDI devices together with a drum machine. The drum machine would be connected to one of the MIDI OUTs, while the synthesisers are connected to the other MIDI OUT.

A second MIDI IN could be incorporated into the design but this would require that the interface be intelligent enough to prevent conflicting access to the RXD line.

Acknowledgement

I’d like to thank Gerry Taylor for his help and advice in implementing the MIDI interface presented here. His help in testing the finished board and proof reading the text of this article is also very much appreciated.

The programs accompanying this article were developed using Manx Aztec C68k v3.60a C compiler, and debugged using the SDB debugger.

Copyright notice

The Amiga MIDI Interface presented in this article is copyright. The circuit diagram may only be used by private individuals on a non-profit making basis. It may not be copied or used for commercial gain without the permission of the author.

Further reading


Listing 1: midi.h

```c
#ifndef MIDI_H
#define MIDI_H

/* midi.h

**************************************************************************

#include <exec/types.h>
#include <kern/wetm.h>
#include <haNvase/intbits.h>

#define TBEn 13 /* transmit buffer empty bit no. */
#define RBF 14 /* receive buffer full bit no. */
#define THB (l << THB) /* transmit buffer high byte */
#define RBH (l << RBH) /* receive buffer high byte */

/* The MIDI baud rate is set to 31250 baud. The equation given on page 199 of ‘Amiga System Programmer’s Guide’ was used to derive the
* `SERV` (serial port initialisation) value given below. It was
determined by experiment, that this value could be as low as 189
* and as high as 111 with the MIDI interface still continue functioning.
* (Well this was true on my MIDI configuration!) */
#define MIDI_BAUD 114

void midi_setup();
void midi_init();
void midi_wait();
#endif
```


**Listing 2: midi_send.c**

```c
#include "midi.h"

/********************************************
* main()
********************************************/
main(argc, argv)
USHORT argc;
USHORT * argv[];
{
    USHORT i;
    USHORT loop;

    if (argc != 2) {
        printf("Usage: send <no-reiterations>");
        exit(0);
    }

    sscanf(argv[1], "%d", &loop);

    midi_init();
    for (i=0; i<loop; i++) {
        midi_send_byte(144); /* note on, channel 1 */
        midi_send_byte(60); /* note no., middle C */
        midi_send_byte(100); /* velocity */
    }
} /* end main */
```

**Listing 3: midi_get.c**

```c
#include "midi.h"

/********************************************
* main()
********************************************/
main()
USHORT b[3];
midi_init();
for (;;) {
    b[0] = midi_get_byte();
    if ((b[0] & 0x00ff) == 0x00ff) {
        printf("goodbye %x\n", b[0]);
        exit();
    }

    if ((b[0] & 0x0080) != 0x80) {
        printf("2 %x\n", b[0]);
        continue;
    }

    b[1] = midi_get_byte();
    b[2] = midi_get_byte();
    printf("%02x %02x %02x\n",
           b[0] & 0xff, b[1] & 0xff, b[2] & 0xff);
}
} /* end main */
```
Listing 4: midi_routs.c

```c
#include "midi.h"

VOID midi_init()
{
    custom.serper = MIDI_BAUD;
    custom.intreq = INTF_RBF;
}

VOID midi_send_byte(b)

VOID midi_get_byte()

VOID midi_wait(mask)

void Solution to the problem posed by John Toebes:

neg.1 D0
sub.1 D1,D1
neg.1 D1
```
Resistors
R1 1kΩ
R2 1kΩ
R3 1kΩ
R4 1kΩ
R5 1kΩ
R6 1kΩ
R7 1kΩ
R8 10kΩ
R9 220Ω
R10 220Ω
R11 220Ω
R12 220Ω
R13 220Ω
R14 4.7kΩ

Semiconductors
D1 1N4148 signal diode
D2 1N4148 signal diode
T1 BC109 transistor
V1 7805C voltage regulator
IC1 6N139 opto-isolator
IC2 74LS05 hex inverter (open collector)

Miscellaneous
3-way Minicon latch
25-way ribbon cable
(max 0.5 metres)

For each MIDI cable
2x 5-pin, 180° DIN plugs
Screened, twisted cable

Connectors
3x 5-pin, 180°, DIN sockets (PCB mounting)
1x 8-pin DIL socket
1x 10-pin DIL socket
1x 25-way IDC PCB mounting socket
1x 25-way IDC cable connector (for PCB socket)
1x 25-way, D-type IDC connector
(Male for A1000 - Female for A500/A2000)

Capacitors
C1 1µF ceramic
C2 0.1µF Minidisc

Figure 4: MIDI interface circuit diagram and parts list
The Art of Assembly Language

18 ways to make your machine code shorter and faster

by John Toebes
Copyright © 1989 John A. Toebes, VIII

John Toebes is a senior systems analyst at SAS Institute Inc. in Cary, North Carolina. He has been programming the Amiga since 1985, and is the author of recent versions of the Lattice C compiler. He is the co-ordinator of The Software Distillery, a group responsible for many popular freely distributable programs including Hack, PopCLf, MemWatch and Blink. John can be reached on Bix (ITOEBES) and on the Software Distillery BBS on (0101) 919-471-6436.

It is often said that if you really want to do something fast, you must do it in machine code. While there is some truth to this, it is more important that in whatever language the code is written, it is written well. With assembly language in particular, it is possible to write bad or inefficient code without even realising it. Only by learning to recognise these bad practices can one hope to learn to write better code.

In a very short period of time, an experienced assembly-language programmer can look at a piece of code and categorise it as either:

1) Great
2) Good
3) Anything else not good enough to be categorised

There aren't any hard and fast rules that one can use to categorise these types of programs, but there are some general guidelines. Although these guidelines tend to vary slightly from person to person, when code is 'good' or 'great' there is rarely disagreement on that categorisation. It is the intention of this article to help you write code that is recognised for its merits.

Great code tends to stand out quite easily - it tends to utilise almost every trick in the book (often inventing a few along the way) and is arranged so that the code happens to fall through to the right areas.

Good code tends to be easy to follow without any truly glaring mistakes or missed optimisations but often loses out on organisational aspects.

On the other hand, bad code tends to be, well... not so good.

We can divide the strategy of writing optimal code into three basic areas:

1. Peephole or instruction optimisations
2. Local Optimisations
3. Global Optimisations

In fact, if these terms seem similar to compiler technology, it is because they are. Compilers have to go through a similar process in making decisions about generating code, although they often don't have as much information available to them as the assembly language programmer does.

Peephole optimisation

In the area of actual instruction selection, the 68000 family offers many opportunities for alternative instructions with its wide range of address modes and instructions. We will focus on the 68000 itself as it is the most common processor, and these optimisations do apply to the later processors. Some guidelines that we can follow in choosing instructions are:

1. Avoid Immediate mode 32-bit values.
2. Avoid 32-bit operands with address registers
3. Avoid 32-bit absolute address modes
4. Avoid MUL/DIV instructions
5. Use the QUICK form of instructions when possible
6. Forget that the CLR instruction exists

Let us look at how we can apply these rules with a few simple instructions:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Value</th>
<th>Length</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1: move.l</td>
<td>#1,00</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>L2: add.l</td>
<td>#2,00</td>
<td>6</td>
<td>16</td>
</tr>
<tr>
<td>L3: move.l</td>
<td>#3,Datavar</td>
<td>10</td>
<td>28</td>
</tr>
<tr>
<td>L4: muls.w</td>
<td>#10,00</td>
<td>4</td>
<td>70</td>
</tr>
<tr>
<td>L5: clr.l</td>
<td>DD</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

Let us look at how we can apply these rules with a few simple instructions:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Value</th>
<th>Length</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1: move.l</td>
<td>#1,00</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>L2: add.l</td>
<td>#2,00</td>
<td>6</td>
<td>16</td>
</tr>
<tr>
<td>L3: move.l</td>
<td>#3,Datavar</td>
<td>10</td>
<td>28</td>
</tr>
<tr>
<td>L4: muls.w</td>
<td>#10,00</td>
<td>4</td>
<td>70</td>
</tr>
<tr>
<td>L5: clr.l</td>
<td>DD</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

These would be coded more effectively as:
As you can see from these simple instructions, there is not always an even trade-off. In example L4, we doubled the code size but were able to cut its execution time in half. Since the code increase is so small, this is a particular situation where it is reasonable to generate more code as the time saving outweighs the increase in size.

With L3, we have an excellent example whereby generating two instructions actually saves both in code size and in execution time.

This brings up an important rule:

7 If you can use MOVEQ for a 32-bit constant - do it!

This applies to any instruction where you can use a scratch data register. For example:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Bytes</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>and.1 #15,00</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>or.1 #2,00</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>sub.1 #28,00</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>cmp.1 #1,00</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Would come out much faster and shorter with:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Bytes</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>moveq #15,00</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>moveq #2,00</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>moveq #28,00</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>moveq #1,00</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Another common area where immediate constants play a role is in stack manipulation. When calling a C routine which requires parameters on the stack, cleaning up from the call generally looks like:

add.1 $12,SP 6 Bytes 16 Cycles

If the number of bytes to add is eight or less, you should instead use:

addq.w $8,SP 2 Bytes 8 Cycles

For nine bytes or more, you can use the LEA instruction:

lea 12(SP),SP 4 Bytes 8 Cycles

Lastly, you should not limit yourself to simple MOVEQ instructions to load constants into registers. Depending upon the constant, there are a number of tricks that you can use to load in a value. For example, to load the value $10000 into a register you might have been tempted to do:

moveq #10,00 2 Bytes 4 Cycles

But wait - you don’t always have to use a register. Sometimes you want to subtract a small number from a register but the number is too big to use for the SUBQ instruction.

Far this, it is quite reasonable to use two SUBQ instructions in a row, so that:

As you can see from these simple instructions, there is not always an even trade-off. In example L4, we doubled the code size but were able to cut its execution time in half. Since the code increase is so small, this is a particular situation where it is reasonable to generate more code as the time saving outweighs the increase in size.

With L3, we have an excellent example whereby generating two instructions actually saves both in code size and in execution time.

This brings up an important rule:

7 If you can use MOVEQ for a 32-bit constant - do it!

This applies to any instruction where you can use a scratch data register. For example:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Bytes</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>moveq #15,D0</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>moveq #2,D0</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>moveq #28,D0</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>moveq #1,D0</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Would come out much faster and shorter with:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Bytes</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>moveq #15,00</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>moveq #2,00</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>moveq #28,00</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>moveq #1,00</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Another common area where immediate constants play a role is in stack manipulation. When calling a C routine which requires parameters on the stack, cleaning up from the call generally looks like:

add.1 $12,SP 6 Bytes 16 Cycles

If the number of bytes to add is eight or less, you should instead use:

addq.w $8,SP 2 Bytes 8 Cycles

For nine bytes or more, you can use the LEA instruction:

lea 12(SP),SP 4 Bytes 8 Cycles

Lastly, you should not limit yourself to simple MOVEQ instructions to load constants into registers. Depending upon the constant, there are a number of tricks that you can use to load in a value. For example, to load the value $10000 into a register you might have been tempted to do:

moveq #10,00 2 Bytes 4 Cycles

But wait - you don’t always have to use a register. Sometimes you want to subtract a small number from a register but the number is too big to use for the SUBQ instruction.

Far this, it is quite reasonable to use two SUBQ instructions in a row, so that:
or for something between $00000000$ and $00000400$, you can use:

```
moveq $540,DD   2 Bytes 4 Cycles
rol.l $n,DD     2 Bytes 8+2n Cycles
```

Given these tricks, in four bytes you can actually generate almost all of the commonly encountered constants where normally you would have needed six bytes. This does add up in the long term.

While these optimisations have all applied to data registers, there are some tricks that apply to address registers:

- Use the .w form of instructions for address registers when possible.
- Do operations in address registers if you can.

For example, if you needed to load a constant value into an address register, you might have coded:

```
move.l $502,A1   6 Bytes 12 Cycles
```

Instead, you can trick the LEA instruction into doing your work for you:

```
lea 502.W,A1    4 Bytes 8 Cycles
```

Of course if you wanted to load a zero into an address register, we can do even better with

```
sub.l AI,A1    2 Bytes 8 Cycles
```

The .w form of the instruction is even more useful for pushing constants onto the stack in preparation for a call to a C routine. Instead of the expected:

```
move.l $1,-(SP) 6 Bytes 20 Cycles
```

or as we know a little better:

```
moveq $1,DO    2 Bytes 4 Cycles
move.l DO,-(SP) 2 Bytes 12 Cycles
```

[total: 4 Bytes 16 Cycles]

We can instead use the PEA instruction:

```
pea 1,w    4 Bytes 16 Cycles
```

For those small constants that MOVEQ can handle, this doesn't seem to be a saving; however, it does save for constants that fit in 16 bits but are too big for MOVEQ. On the 68020 and the 68030, the MOVEQ sequence is actually faster for the constants due to pipelining in the processors, so you are probably better off using it when you can.

On the Amiga, one address register manipulation that occurs frequently is converting a BPTR into an APTR. Since a BPTR is shifted right two bits, one might be tempted to use a data register to perform the operation (since shift and rotate instructions do not operate on the address registers).

Even transferring to a data register is expensive:

```
exg   A0,DD    2 Bytes 6 Cycles
lb1.l $2,DD   2 Bytes 12 Cycles
exg   DD,A0    2 Bytes 6 Cycles
```

[total: 6 Bytes 24 Cycles]

Using a MOVE instead of an EXG instruction only saves us four cycles. We can do much better by using two instructions instead:

```
add.l AD,AD   2 Bytes 8 Cycles
add.l AD,AD   2 Bytes 8 Cycles
```

[total: 4 Bytes 16 Cycles]

Certainly there are many other peephole optimisations available on the 68000. However, these are the most prevalent and useful for typical code.

An excellent method of learning them is to sit down with a good Motorola manual and read the timings. Some of them are actually quite surprising.

However, these optimisations by themselves are only a small part of what makes an assembly language good (or great as we would like it to be). In reality, you can get much more of a gain by making algorithmic changes to your code than by making simple instruction substitutions.

For this reason, we now examine the range of local optimisations available.

**Local optimisations**

Unlike the peephole type of optimisations, local optimisations require a little more thought in order to arrange, and often have minor effects on the algorithm you are trying to implement. The simplest that comes to mind is in operations which would correspond to a multiplication in a high level language.

```
move.l index,DO
muls.w #4,DO
```

You would code:

```
move.l index,DO
lsll.l #3,DO
```
On the subject of shifts, there are some simple rules to follow:

10a If the number to shift is greater than 15, do a SWAP instruction and then subtract 16 from the number to shift.

10b If the number is 8 or more, generate a shift by a constant 8 and then subtract 8 from the number to shift.

10c If the number is 1 and the shift is left, generate an ADD instruction instead.

10d If the number is greater than 0, generate a shift instruction for that amount.

Taking these rules, we get the following sequences:

```
add.1 D0,D0 ; shift left by 1
ls1.1 #2,D0 ; shift left by 2
ls1.1 #4,D0 ; shift left by 4
ls1.1 #8,D0 ; shift left by 8
ls1.1 #8,D0 ; shift left by 9
add.1 D0,D0
ls1.1 #6,D0 ; shift left by 10
ls1.1 #2,D0
swap D0 ; shift left by 16
c1r.w D0
swap D0 ; shift left by 24
c1r.w D0
ls1.1 #8,D0
swap D0 ; shift left by 25
c1r.w D0
ls1.1 #8,D0
add.1 D0,D0
lsr.l #1,D0 ; shift right 1 (no optimise)
c1r.w D0 ; shift right by 16
swap D0
c1r.w D0 ; shift right by 25
swap D0
lsr.l #8,D0
lsr.l #8,D0
```

All of these sequences turn out to be better for the 68000 than loading a shift constant into a register and doing the shift, as they use one less register and are all faster. This particularly true for the 9-15 range. When generating the 16 case, always remember to do the swap in the correct place.

It is important to note that these shifts really aren't very careful with the sign bit, but in assembler this is rarely important. If you do have to handle a signed shift of 16 or more, you can use the EXTL instruction instead of the CLR.W instruction.

One particularly fancy optimisation that is useful when writing an assembler in assembler takes advantage of forcing bits out as the shift is being done. In generating the register target field of an instruction, for instance, it is necessary to mask the value with a 7 and then shift it left 9. A good assembler programmer might have coded (assuming the value was already in D0)

```
moveq #7,D0 12 Bytes 4 Cycles
and.1 D1,D0 2 Bytes 8 Cycles
ls1.1 #8,D0 2 Bytes 24 Cycles
add.1 D0,D0 2 Bytes 8 Cycles
```

But a more astute programmer would have noticed an opportunity here (knowing that the register value will fit into a byte) and coded instead:

```
ls1.b #5,D0 2 Bytes 16 Cycles
ls1.1 #4,D0 2 Bytes 16 Cycles
```

11 Track condition codes with a vengeance

A great assembly program almost never has a TST instruction in it. The 68000 is very good about setting condition codes to reflect almost every operation. Only when you are putting something into an address register is the condition code left unchanged.

However, this can bite you if you simply add code to an existing program without checking the condition codes, as almost anything can destroy them. It can be more beneficial to remember the MOVEM instruction which, although it is larger and slower than the MOVE instruction, can safely load or store a single register without affecting the condition codes.

It is a good rule of thumb to forget the TST instruction (if possible) and arrange your code so that the condition codes fall through.

Condition code tracking is extremely useful for multi-way branching. Assembly language programmers are afforded the simple luxury of testing for three things at once:

```
add vall,D0 ; do some operation to...
            ;...set the CC (not a TST)
blt.s negative ; if negative, we need...
            ;...to do some more
beq.s done ; at zero, we are complete...
            ;...or else we come through...
            ;...with a positive value
```

In most high level languages (with the exception of Fortran) this type of situation is almost never seen, but in good assembly programs, values and flags are arranged so that this situation can be used.

When choosing the branch instructions, it is important to be aware of how often the branches are likely to be taken. The branch condition should be such that you know that it will not be taken the majority of the time.
Detection of error situations should be done by a test that branches out of line, thus allowing the normal case to continue past the branch instruction. In this way, on the higher processors, there is no pipeline break, and even on the 68000 you take advantage of the shorter cycle time for not taking the branch:

```
moveq $1, D0
and.l D0, D1 ; is the error request..  
             ; ...flag set?
bne.s error1 ; yes so go tell them..  
             ; ..about the error
             ; fall through here in the most common case
```

12 Abuse the DBcc instructions

The best instruction of all, however, is the DBcc series of instructions. Even on the 68010, this puts the processor into a mode where the instruction covered by the loop is kept in a cache for fast, repeated execution.

However, the DBRA instruction has some very important drawbacks to be aware of:

12a The loop counter is only 16 bits. The high order portion of the register is untouched.

12b For performance, you should only include a single instruction in the loop.

12c Loops entered at the top are zero-based, executing one more time than the number in the register.

Even with these limitations, the DBRA instruction can be used in many situations. The most common is in copying areas of memory:

```
moveq $15, D0 ; copy of 16 bytes of..  
             ; .. memory on any boundary
move.l source, A0
move.l dest, Al
lab: move.b (A0)+, (Al)+
dbra D0, lab
```

If you don’t know the length ahead of time, this is easily remedied by entering the loop at the DBRA instruction:

```
move.w len, D0
move.l string, A0
moveq $char, D1 ; character to search for
bra.s dbra
lab: cmp.b (A0)+, D1
dbe.s D0, lab
```

Now in this case we had to be careful. The length could have been zero, so we need to start at the bottom of the loop. However, the loop would have terminated if the condition code had been EQ, so we make sure that the last thing we do before starting the loop actually sets the CC to a non-EQ value - hence the MOVEQ before the BRA.

Global optimisations

These peephole and local optimisations allow you to fine tune hot spots in the code and in general make the microscopic cases run faster. However, for the most return on effort, there are several global optimisations that are a winner:

13 Take time to choose registers carefully

A good program attempts to assign registers to the high traffic items, or even to the most important at a particular instant. The secret to a great program is arranging the code and register choices so that the value needed is always in the right register. In particular it is desirable to avoid having to do a MOVE register to register instruction just to get something in place for a call.

14 Avoid MOVEMS when possible

Because the standard 68000 calling conventions require you to preserve all registers except D0/D1/A0/A1, it is necessary to save any other destroyed registers on entry and restore them on exit. It is most desirable to make register selections so that as few registers as possible are saved. In fact, if you only need to save a single register, then substitute a MOVE instruction instead.

15 Avoid LINK and UNLK instructions

While LINK is often used in high level languages, its use in assembly programming is actually quite rare and mainly used for creating temporary storage for items too large to fit in a register. If there aren’t enough registers to go round for an operation, thus requiring the storage of some on the stack, it may be better to create a subroutine for part of the operation (if time is not critical) or even to push the parameters on the stack instead of storing at a fixed location in an allocated frame.
16 Make intelligent choices for flags and values

If you have three possible states for a flag, then choose -1, 0 and 1 so that you can use a three-way branch. If the value will be used to index a table, choose even ones that correspond directly to the entry index in order to avoid a multiply. For bits, put the most commonly tested bit into the sign position so that the single bit gets set as a side effect of loading the flags. With a long, you have 3 bits that can set the sign bit based on the word length - pick these first.

17 Use tables when appropriate

Many operations can be readily done with tables. For example if you are doing a transformation for on-screen rotations, it may be sufficient to pre-calculate the sine for all 360 degrees, round off the rotation angle desired to the nearest degree and then look it up directly instead of going through an expensive calculation. Jump tables to perform operations (especially when the number for the operation is intelligently selected as above) are particularly quick.

The heavy artillery

By oo means is this a complete list - the number of opportunities for optimisation on the 68000 are endless. However the amount of time to perform the optimisations is finite. For this reason, it is necessary to weigh the importance of a particular optimisation against the time necessary to implement it.

One excellent tool to consider using is a profiler. This will point out where in the code the most time is being spent and allow you to concentrate on that area.

A good rule of thumb is that 95% of the time will be spent in about 5% of the code. There is little benefit in optimising the remaining 95% if you haven’t paid attention to the hot 5%.

A parting trick

18 Learn how to use the X bit

The X bit is one of the strangest bits in all the 68000 architecture. It is set by very few instructions, but does have the property of allowing you to set an entire long word to 0 or -1 based on its value:

\[ \text{subx} \text{.} \text{X} \text{,} \text{Y} \text{,} \text{Z} \] \[ \text{subx} . \text{X} \text{,} \text{Y} \text{,} \text{Z} \]

With this in mind, test your ability to optimise 68000 code by figuring out the best sequence of instructions for the following: Given a value in D0 and with condition codes in an unknown state, set D1 to 0 if D0 currently contains a 0 and to 1 otherwise. You may destroy D0 if necessary. The answer is lurking on page 36.

---

WHITE'S COMPUTERS

37 DAPHNE AVENUE, CASTLE HILL 2154

(02) 634 6636

Quality computer supplies at reasonable cost
- Blank disks
- Computer paper
- External floppy disk drives
- Hard disk systems for A500 and A1000
- Printers
- Ribbons, and
- Most other consumable items
- Star Cursor joystick, 100% Australian made—3 year warranty

Tuition classes for beginners
Bookings essential
Contact Chris or Alan White

Bi-monthly Amiga meetings are held at the Girl Guides Hall, Bounty Avenue, Castle Hill, between 12 noon and 4 p.m. Dates for 1989 are 6 August, 1 October, 3 December. Everyone welcome. Commercial and non-commercial displays. Accessories on sale. If you have developed a special expertise or application on the Amiga and would be prepared to demonstrate it for us, please call.

Latest magazines in stock include
- Transactor for the Amiga
- Amiga User
- Amiga World (flown in monthly)

MAIL/PHONE ORDERS WELCOME

■ BANKCARD ■ MASTERCARD ■ VISA
Hard Disk Backup

Three commercial products to bring you peace of mind

by Steve Ahlstrom
Copyright © 1989 Steve Ahlstrom

One of the fastest growing segments of the Amiga peripheral market is the hard disk. Until recently there were only a couple of manufacturers of Amiga hard disk controller cards, now there are so many one can lose count!

A hard drive on an Amiga transforms the machine. If you do not own one, you may be under the impression that you do not need one - if you only knew. With a hard drive addition, it's like getting a brand new computer.

Buying a hard disk solves problems. No more looking through stacks of disks for the program you want to run and, if you are like me, half or more of these are unlabelled. The increase of speed in reading and writing is an appreciated pleasure in itself.

However, owning a hard disk is not all roses; there are a few thorns thrown in. The biggest of them all is how to make sure all that software you've put on your hard disk will survive if you should have a sector go bad ... or maybe turn your computer off during a write operation ... or a half dozen other things that might go wrong.

I have reviewed 3 different hard disk backup programs. The first program is the best known. Second is one of the earliest for the Amiga. The third is a brand new entry into the marketplace. They all have strengths and weaknesses. I am going to be up front and tell you that, in my opinion, no one is better than the others. Read on and find the one that meets your needs.

Quarterback

Quarterback's biggest advantage is that it will backup any AmigaDOS device, which includes Bernoulli boxes, streaming tape devices, high density floppy drives, jh0: - essentially all of those non-trackdisk.device peripherals.

Using Quarterback is simple. It boots up into a full sized workbench window with 2 gadgets showing, Backup Files and Restore Files. By clicking on Backup Files you are prompted for the hard disk device name and optional subdirectory path. You are also prompted for the primary and alternate destination devices. If you wish, Quarterback will backup on 2 floppies, which cuts down on overall backup time.

When the backup to your primary device is completed, Quarterback will automatically start writing to the secondary device. In the meantime, you put in a new disk into the primary device. This way Quarterback never skips a beat.

Quarterback will build a list of file names in your backup path. The building of this list is what gives Quarterback some rather impressive timings later as you actually do the backup. Quarterback takes the time to create the list before it starts the backup. ExpressCopy and Saf-T-Net build the list on the fly. This really needs to be taken into consideration if your primary concern is speed of backup. It's all relative - time is time, whether that time is taken before or during the backup phase. Do be wary of benchmarks based only on the time needed to write data to a disk.

After the building of the list, Quarterback will show you the number of files it will backup, total amount of bytes in those files, and an estimate of the number of floppies needed. At this point there are various options to select for including or excluding files by name, date, whether the archive bit is set, or all. You can select a subdirectory in your backup path and apply the inclusion/exclusion features to the subdirectory rather than the entire backup path. This allows for flexible control over the data you want to backup.

Next, you are presented with various options to select - write with verify, overwrite AmigaDOS disks, keep between disks, set archive bits, and so on. The most important of these options is the AmigaDOS disk warning and the write with verify features.

The overwrite feature, in its default mode, will check if the inserted floppy is a valid AmigaDOS disk. Should a disk be inserted that you do not want to overwrite, a warning will be provided. This is a nice safety feature. You are given the option of continuing or inserting a new disk. When this option is turned off, Quarterback does not check.

The other important option is the creation of your backup with, or without, write verify. With verify, Quarterback reads
Serf-T-Net was one of the earliest entries into the Amiga hard disk backup market. It has been plagued by poor marketing and has not received the attention it should have. It was initially marketed by RSN Software who licensed the program for distribution. They were a bootstrap operation that ran up against a Catch-22 situation— they could not afford to advertise unless sales were strong, and sales had no chance of being strong if no-one knew about the program. Eventually, RSN decided its experiment in the Amiga marketplace was a failure. The marketing rights to Saf-T-Net reverted back to its author, Bob Rakosky.

The future of Saf-T-Net looks positive. It has recently been given a face-lift with many new features added. Metadigm (of MetaScope fame) has licensed the product for distribution and this Metadigm version should be on your dealer's shelves about the time you read this.

When you bring up Saf-T-Net you are presented with a custom screen full of gadgets. From here you choose all of your options. And there are a lot of them.

On one screen you choose backup or restore, full name of the device/directory to backup, whether you want a full backup, incremental backup, restore, and so on.

After you have set the various options wanted, click on OK. You are shown a new screen— this is where all the action takes place. You start your backup/restore from here and are shown the full path names as the process occurs. Should you get to this screen then decide the options previously set are not the ones you want, you can back-out and return to the configuration screen.

Unlike other programs, when you want to do an incremental backup, instead of creating a new backup set, you simply add to your existing backup set. This may be a minor point but it is convenient. Your backup disks are numbered by the program. When you want to create an incremental backup Saf-T-Net asks you for the last disk in your backup set. It then reads in the cataloguing information, and starts writing to that disk (if there is space).

It is suggested that a full backup should be done once a month, and an incremental backup at least weekly. With this approach you only have one set of backup disks to maintain. With other programs you have at least two, and sometimes many, separate sets of backup disks.

Backing up with Saf-T-Net is not as speedy as QuarterBack. Saf-T-Net does not offer a verify/no verify mode, it only verifies. Should a disk error occur while writing to the backup disk, Saf-T-Net will attempt to format the track, verify it, and continue its operation.

Saf-T-Net does not keep its catalogue of files on your backup disks. This can be saved anywhere you like which gives you an extra bit of insurance should one or more backup disks go bad. As mentioned earlier, Quarterback keeps this catalogue on the first and last disk of the backup set. While it is unlikely that both the first and last disk would go bad, if they do, your backup set is worthless.

You can't restore without the catalogue. With Saf-T-Net, you can write it out to a floppy or to hard disk. Writing it out to
hard disk is not recommended because if the hard disk develops problems you could lose your catalogue. However, using this approach, allows you to keep more than one copy of the catalogue, depending on your state of preparedness (or paranoia). Even without the catalogue, you can still restore from your backup disks. Also, a new catalogue can be created from your backup set.

Saf-T-Net does not write in AmigaDos format. It packs 907.5k of data on a diskette and does use compression on some files, but the type used really is not that significant. It will give a 5-10% compression over the entire backup set. Compression is time intensive so the author only uses it when there would be little or no time penalty.

Unlike Quarterback, Saf-T-Net backup disks cannot be copied with the AmigaDos Diskcopy program but a program called Saf-T-Copy is included which will allow you to copy your backup set.

Saf-T-Net is a very solid program. It is not the fastest nor the most feature laden, but it gets the job done. Some of the unique features is does have are there for user safety. It’s main goal is to help protect your data.

Saf-T-Net is not copy protected, runs in 512k, is fully multitasking, and has a suggested retail price of $49.95. It is available from your local dealer or from:

Metadigm, Inc.
19762 MacArthur Blvd.
Suite 300
Irvine, CA 92715

ExpressCopy

ExpressCopy (XCO) is the new kid on the block. It is feature packed. Written by Glenn Nielsen of ExpressWay Software, it is sure to become a strong contender.

The most obvious difference with XCO is that it writes AmigaDos format files. Should you want to use the new 1.3 Fast File System (FFS) on your floppies, XCO will accommodate you.

Using FFS on floppies is not approved by CAM, but it will give you a nominal increase in both access speed and storage capacity. Normally, to use FFS on floppies you must create a mountlist entry and mount your floppy (an example mountlist is provided in the XCO manual). A diskechange command must be issued after inserting a new FFS formatted floppy. AmigaDos does not automatically recognise it.

While running XCO, should you decide you want to use FFS formatted floppy, XCO will recognise it. XCO goes so far as to format the disks in FFS format for you while you are creating your backup - even if you don’t have the mountlist entry or haven’t mounted the FFS floppy! Of course, to restore from these FFS disks you must set up the mountlist and mount the drive.

By creating AmigaDos files, XCO has advantages over Quarterback and Saf-T-Net, although there are some disadvantages.

On the pro side, the backup disks that XCO creates have the same directory paths and files as the source directory. Should the hard disk fail for some reason, the files backed up using XCO could be useable from floppy.

Imagine this panic situation: your hard disk becomes invalidated and nothing you do will revive it short of formatting it. So, you format it with the idea of restoring from your backup set. Half-way through the restore you come up with a bad floppy. What do you do? With either Quarterback or Saf-T-Net you are pretty much out of luck; you will lose the files (unless you have a backup of your backup set!). With XCO, you stand an excellent chance of restoring the disk with either DiskSalv or DiskDoctor. Using AmigaDos format increases the security and integrity of your backup.

You can use AmigaDos or your favourite directory utility to restore. However, XCO comes with a separate restore program although the only time you really need to use this program is if you have backed up files that are larger than the capacity of the floppy. Restoring with AmigaDos will not help you here.

Using AmigaDos format does have its downside. While many might think that it would take longer to create AmigaDos files, that’s not the case. XCO is very competitive time-wise (see the chart below). The main disadvantage is that you store less data per disk. Quarterback writes 901k to each disk. Saf-T-Net writes 907.5k to each disk. With XCO, it’s variable. At best, it can write a bit over 800k per disk, sometimes much less. If you had filled a disk to 500k, there would still be a little over 300k of data space free. If the next file to back up is 400k in size, XCO will ask for another disk in the set and effectively leave the previous disk only partially filled. Depending on the luck of the draw for this case, the distribution of various sized files on your hard disk, XCO will require more disks for the backup than the others.

Even if every disk is filled to capacity, XCO will still use more disks because each disk filled to capacity holds less data. ExpressWay has said that in a future update they will attempt to utilise the wasted space by holding back the write of the large file and write other smaller files in the directory. They intend to save the writing of the larger file until either the directory has been backed up completely or until it is time for the next disk to be inserted.

Like the others, XCO has a full complement of options for backing up and restoring. You can include/exclude files by the archive bit, date stamp, and/or filename pattern matching.

XCO will automatically detect if the disk inserted is a valid AmigaDos disk. If so, a warning will be given and you will be asked if you want to write over it. This helps protect from inserting a disk you do not want to lose; such as one of the disks you just completed in your backup.
XCO gives you a lot of control in the use of the program. It is fully multi-tasking. Should other programs be active while running XCO, you can easily change the priority level. Should you need to use the same devices XCO is using, you can click on the Halt button and XCO will stop after it has completed writing the current file and let you use the drive for another program. When done, click on Resume and XCO will check that you have re-inserted the correct disk or disks and resume the backup.

If XCO comes across a file or directory that it cannot access because another program has a lock on it, you have the option to complete the task holding the lock and have XCO access the file/directory again, or skip the files completely.

As with all hard disk backup programs, integrity of your data is of prime importance - bells and whistles come second. XCO allows you to generate a listing file of all files copied which will contain not only the file names but also all of the file attributes and a CRC checksum value for each entry. You can then use the (supplied) program XCOcrc, from the CLI, to check the integrity of the files XCO wrote.

XCO will use up to 4 disk drives to write the backup. This is a real time saver since you can pre-load the drives with disks. XCO can also use up to 4 drives to write 4 copies of the backup. In other words, it will write the same data to all 4 destination drives.

With this, XCO becomes a valuable tool to people who don't have a hard disk. If there is a need to make multiple copies of a disk quickly, you can set up XCO to use a floppy as the source and write to multiple destination devices. This would be a real time saver for someone such as a user group librarian, or anyone who has a need to make multiple copies of a disk.

ExpressCopy is not copy protected, runs in 512k, is fully multitasking, and has a suggested retail price of $44.95. It is available from your local dealer or from:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Quarterback</th>
<th>Saf-T-Net</th>
<th>ExpressCopy</th>
</tr>
</thead>
<tbody>
<tr>
<td>AmigaDOS Format</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>No verify option</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Non-trackdisk support</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Report Generation</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Backup files larger than disk</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Write a disk - No Verify</td>
<td>57 secs</td>
<td>n/a</td>
<td>48 secs</td>
</tr>
<tr>
<td>Write a disk - Verify</td>
<td>1:46</td>
<td>2:21</td>
<td>1:28</td>
</tr>
<tr>
<td>FFS floppy support</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Multiple floppy support</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Multiple copy support</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Format disks while running</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Runs in 512k</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Copy protected</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Suggested Retail Price</td>
<td>$69.95</td>
<td>$49.9</td>
<td>$44.95</td>
</tr>
</tbody>
</table>

Summary

Which is the best? It all depends on your needs.

All of the programs claim to be fast. Fast is relative. With verify turned off, Quarterback and XCO will write a disk in less than a minute. Performing a hard disk backup without verifying the writes, in my opinion, is a waste of time. You would not be doing the backup unless your data was important and not easily re-creatable. By not verifying the write to your destination you have no guarantee that the data you are saving is valid.

When you do use the verify write option, the backup time increases considerably. XCO and Quarterback are about even in speed. XCO is faster but writes less data to each disk. It evens out over the entire backup process. Saf-T-Net, while writing more data to the disk, is considerably slower.

If you need to backup to/from non-trackdisk devices, you need Quarterback. If you need to the backup disks to be in AmigaDOS format XCO is the choice.

If you want to utilise the most capacity on your backup disks with the added security of placing the file catalogue anywhere you want, it's Saf-T-Net.

Whatever you decide, don't find yourself with a hard drive that suddenly crashed without a backup!

For the timing tests I used a directory which contained an equal number of files 5-50k, 50-100k, 100-300k in length. Testing was done on an A2000 with an A2090 and a MiniScribe 8425F (40 ms) hard drive.
ARexx Programming

Part 2 - Decision making and loops

by John Carpenter

The Rexx instruction set is small compared with some languages, but what there is is supplemented by a large set of functions (a concept familiar to C programmers).

Rexx beginners often get confused with how Rexx differentiates between instructions, variables and functions.

Functions are always suffixed with parenthesis and normally return a value, such as in \l 1 = Length(string). In this case \l is a variable which is assigned the length of the string contained in variable string.

An instruction is a keyword that is not followed by a colon or an equal sign. If the keyword is followed by a colon then it is a label, if it is followed by an equal sign then it is a variable.

It therefore follows that there are no reserved keywords as such, but using instruction keywords as variables or labels is likely to create a great deal of confusion for the reader.

Take the following program for instance (note that the line numbers in the examples are for easy reference, and are not part of the programs):

```
1: /* Rexx programs must begin with the /* */ */
2: Say = "Say"
3: Say Say
4: Exit 0
```

The result of this program will be the word "Say" being displayed. On the second line, Say is a variable while on line 3, the first Say is the instruction whilst the second is the variable again.

The first instruction we came across was SAY. The syntax is:

```
SAY [expression]
```

The square brackets indicate that the contents are optional. SAY on its own will cause a newline to be sent to the console. The expression can contain as many strings, variables and functions as you like. If you cannot fit the expression on one line then a comma may be used to indicate that the clause continues on the next line.

For instance:

```
1: /* */
2: Say "Hello there." "What a",
3: "wonderful language Rexx is."
4: Exit 0
```

The result of executing this program is:

```
Hello there. What a wonderful language Rexx is.
```

What may surprise you about the output from this program is that there is a blank between there and What. Given this fact, how do you join together strings or variables without blanks?

```
1: /* */
2: Say "This won’t work""as it includes double quotes"
3: RC=0 /* set up a variable */
4: Say 'This will, return code='+RC
5: Say "So will th"||"is."
6: Exit 0
```

Line 2 results in a double quote appearing in the middle of the output. Had the string been delimited by single quotes, the word won’t would need to be coded as won’lt.

The fourth line does not have this problem, so butting the variable RC up to the end of the string produces the desired effect. The string concatenation operator (||) could have been used as in line 5.

Decision making

The IF..THEN..ELSE sequence is similar to that of BASIC and its syntax is:

```
IF expression THEN [statement]
```

If the expression is true then the statement will be executed. The THEN statement does not need to be on the same line as the IF. If you wish to execute a null clause, use the NOP code (no-operation).
Line 3 executes a no-op while line 4 has the real statement. This technique avoids having to use inverse logic that often leads to errors.

Line 5 shows that an ELSE clause is not essential.

Variations on DO

To execute a group of statements, rather than the individual statements we have seen so far, we need to use the DO..END instructions.

DO indicates the beginning of a block and the END instruction is the block delimiter.

Both lines 6 and 7 will be executed as a block in this example.

The next DO..END block demonstrates the BREAK instruction. BREAK is used to break out of a block prematurely.

As the program stands, both SAYS will be executed; but if a was equal to 1 then the first SAY would be executed and then the BREAK would be executed. This would bypass the second SAY and resume execution after the END statement - the EXIT instruction.

The DO..END pair is not only used to delimit a block of instructions but can also be used to control a loop. BASIC, of course, has the FOR. NEXT and FOR.NEXT.STEP loop constructions. Rexx has an equivalent construction in the form of a variation on the DO loop. This is DO TO and DO TO..BY and its syntax is:

```
DO var=expression TO expression [BY expression]
```

Take the following program:

```
1: /* */
2: a=0
3: Do While a<3
4:   Say "While"
5:   a=a+1
6: End
7: Do Until a=3
8:   Say "Until"
9: End
10 Exit 0
```

The result is While gets printed three times and Until once.
Looping forever

The final DO..END variant is DO FOREVER. This sounds like a programmer's nightmare but in actual fact it is very useful. Control of the loop is achieved by the LEAVE instruction. When the LEAVE instruction is executed then clause following the END statement is executed. The syntaxes are:

```
DO FOREVER
LEAVE [variable]
```

If LEAVE is executed without the variable name then it will exit the current DO loop. The LEAVE instruction can be used with any DO..END loop. With the variable name specified, the LEAVE instruction can exit nested loops.

```
1: /* */
2: Do i=1 To 3
3:   Do j=1 to 5
4:     Say i "times" j "equals" i*j
5:     If i=3 & j=3 then Leave i
6:   End
7: End
8: Exit 0
```

The result is the 11 to 15 and the 21 to 25 tables - the 3 times table terminates at 33. At this time the LEAVE instruction causes the instruction following the outer DO loop's END statement to be executed - the EXIT instruction.

The other loop control instruction is ITERATE.

```
1: /* */
2: Parse Upper Arg FILE /* get file description */
3: If Open(' file' ,FILE,'R') then nop /* open file */
4: else do
5:   Say 'Unable to open file' FILE
6:   Exit 1
7: end
8: Do forever /* read loop */
9:   Rec = Readln('file') /* read a line */
10:  If Eof('file') then leave /* if end of file then leave loop */
11:   Interpret Rec /* execute expression in Rec */
12:  end /* end read loop */
13: a = Close('file')
14: Exit 0
```

If we create an ASCII file, TEST, containing the following:

```
Say 'Hello there'
a = 'Say "nested interprets"'
Interpret a /* this also works */
```

And then execute the program, we will get:

```
The cat sat on the mat
10 Mar 1989 TODAY
Hello there
nested interprets
```

The line with the ITERATE instruction is interesting as it contains a nested function.

```
Upper(Substr(Rec,1,1)) initially obtains the first character from variable Rec and then converts the result into upper case.
```

The ARG instruction

In the previous example, ARG is used to get the arguments passed to the program using the form:

```
Parse Upper Arg FILE
```

This is, in fact, three instructions: namely PARSE, UPPER and ARG. If you check out the manual you will realise that I could have just written ARG. In this particular instruction, all the arguments would be passed into the variable FILE.

INTERPRETing expressions

The INTERPRET instruction is rather novel. Its syntax is:

```
INTERPRET expression
```

The expression is first evaluated and then executed. If the expression contains one or more variables then these are evaluated for execution.

```
1: /* */
2: a = 'Say "The cat sat on the mat"; Say Date()'/*nest the interprets*/
3: Interpret a 'today'
4: Drop a
5: If Open('file','TEST','R') then nop /* open file */
6: else do
7:   Say 'Unable to open file' FILE
8:   Exit 1
9: end
10: Do forever /* read loop */
11:   Rec = Readln('file') /* read a line */
12:   If Eof('file') then leave /* if end of file then leave loop */
13:   Interpret Rec /* execute expression in Rec */
14:   end /* end read loop */
15: a = Close('file')
16: Exit 0
```

If we create an ASCII file, TEST, containing the following:

```
Say 'Hello there'
a = 'Say "nested interprets"'
Interpret a /* this also works */
```

And then execute the program, we will get:

```
The cat sat on the mat
10 Mar 1989 TODAY
Hello there
nested interprets
```

Next time, I will deal with PARSING, and how to call other programs and functions.
Random Access

User Group Secretaries: Did you know that we offer a special deal on this magazine to User Groups who buy 10+ copies? No? Ring and find out more!

Direct Source Software: Two local businesses now offer software at direct sale prices. Interlink Software Pty Ltd were first. They can be contacted at PO Box 1155, Tuggeranong ACT 2900 or ring (062) 31 0155 9-5 weekdays. They offer a wide range of Amiga/C64/C128 software and books, and accept all normal methods of payment. Plastic money is by mail only, unless a prior arrangement has been made.

Briwall Australia reports brisk business, especially in the (now) much ignored C128 area. They are at PO Box 9, Rivett ACT 2611. Their phone is (062) 88 0131 or fax (062) 88 0337. They too have a wide range of Amiga/C64/C128 software and books, plus things like joysticks, mousepads, ribbons and so on.

Both promise prompt return service, and by all accounts, keep their respective words.

Word Processors: My trials and tribulations have been boring my friends for a few months now. After hacking through a dozen packages that did little or nothing for me (or for the credibility of those who wrote the accursed things) I decided to settle down and really get into Word Perfect.

After spending 4 hours setting up the package, my temper was not cool. But I kept on, and found it to be very good, if really for the professional user. Then came news that there were to be no more updates, and my interest waned rapidly.

It was about then that I discovered Uedit—a shareware package that was so easy and flexible to use that I hastened to send money to Rick Stiles in the USA.

Let me wax lyrical for a bit. Uedit is not a WYSIWYG program, so it is faster, easier to use and more comprehensive. It gives you power, speed and wide ranging flexibility. It can mimic Wordstar, or be a disk utility. In between times, it processes words well. Multiple buffers allow you to compile a document from a selection of other documents, simply by cutting from one buffer into another. You could have 100 files in memory at once if you chose! If you don't like the way it does things, you can redefine commands, keys and buffers to suit yourself. And it multi-tasks beautifully.

Uedit comes in two forms—UES (S = short? simple?) and UE. For the most part, UES is more than adequate. But if you want to customize UES, you need UE. You can easily make UE into the most powerful editor you have ever used. This nirvana can be yours for a song. UES is US$20, UE is US$50 (plus US$8 for airmail delivery). Where? OK—Rick Stiles, PO Box 666, Washington, Indiana 47501 USA. Tell him Paul sent you.

The Commodore A590 hard disk/memory expander for the A500 is reportedly due to go on sale soon. I asked Commodore for a review model, but nothing was forthcoming. So all I can say is that, for around (my guess) $800-$900 you can lash a 20 Meg hard drive to your A500 and do away with the floppy blues. If you buy the memory chips ($250/Meg ???) you can have 3 Meg and a hard disk in your A500 for under $1500. Ergonomically the unit will make the Amiga kind of ugly, and generally get in the way of a second external drive (A1010), but that could probably be dispensed with anyway. Going the A590 way will give you a very comprehensive version of the Amiga at modest (not low) cost, and could cut into sales of the larger models???

Project D: Looking around for a disk cataloguer, I came upon Project D, written by Ben Fuller of Mesa, Arizona. The disk also contains a powerful copy program, an Atari/MS-DOS copier, and a disk examination utility.

I catalogued up some disks, then chose the multi-column print option. Amiga disks tend to hold a lot of files, and 20 disks gave me nearly 1600 files. So the economy of multi-column output was welcome.

Darn it all if the program didn't guru every time. The guru message (#3—software) wasn't too illuminating. So I redid the catalogue, down to 1200 files. Same result.

I have no clue as to why that particular printer (or disk) output goes haywire. Maybe someone has had a similar experience, and could help solve the mystery. Meantime, I'll retire the disk to the back of the box, to keep it from harm.

On the subject of odd things (leave me out!), why does DiskMaster always think its tomorrow? Copy some files on an Amiga with a clock, and look at the results in the datestamp field.

OKI printers: A new 9-pin dot matrix printer is now available for the Amiga. The Microline-ML172 can pound along at 180 cps in draft mode, 30 cps in NLQ mode. Retailing for under $400, this sort of printer would be a good general purpose unit for everyday use. At a resolution of 240x216, it will produce reasonable graphics. If you can't see one at a dealer near your home, ring IPL-Datron Pty Ltd on (02) 699 4824 or fax (02) 698 4043. They also have offices in Melbourne (03) 690 9855 and Brisbane (07) 368 3292.

Locals: This seems to be Canberra week! Craig Fisher is a young man going places quickly. Although Commodore has taken no notice of his skills, his work is starting to be noticed. Starting with MIDI manufacture way back when the Amiga was new, he has progressed to SMARTKEY and CALCKEY. We will include samples of his work on an Amiga disk soon. Craig's phone number is (062) 92 3115.

Transactor in Australia is pretty much a one-man band: me. To keep costs (hence cover price) down for your benefit, I do most everything myself. So if I'm not home when you call, and my family answers, please be kind to them. They give me great support and encouragement to keep the magazine coming along. Just tell them where you can be contacted, and I will get back to you as soon as possible.

Thanks.

Paul Blair
HDTutil  The complete Hard Disk utility  $69.95

Fast backup mode - 28 Meg. in under 38 minutes @ Include/exclude files by wild card - date - archive bit - list - globally - or by point and click @ Auto configures to multi hard drive or partition systems @ Works with all AmigaDos compatible hard drives @ Multi tasking - designed to run in the background @ 3 backup/store modes - Dos backup mode allows you to access your backup disks with standard Dos tools @ Formats disks as required @ Catalog of files @ Build, save and manipulate all files and directories/subdirectories @ Runs from WBench or CLI @ Works equally well with dual floppy @ Full intuition interface for ease of use @ Many additional features allow you to - copy files - delete/invert/renam files or directories - make new directories - copy or format floppy disks - interface with other programs to TYPE or EDIT text files, View picture files and more @ Written in 68000 Assembler code @ Minimum 512k Amiga required @ NOT COPY PROTECTED

AutoKick Modify 1000 KickStart disks  $29.95

(not the 500/2000 with additional equipment, burn new Eproms)
Software automatically contiguous blocks of fastmem - increase available chipram @ Perfrom internal 20k of another block of fastmem to speed reboots @ Install a font of your choice inplace of Topaz-8 font provided on KickStart @ Install code into KickStart to prevent the spread of any of the known viruses @ Reduce the annoying click from empty drives @ Speed up floppy operation @ Modify KickStart to use any floppy to boot from @ Full intuition interface - all operations selectable from gadgets @ Runs from CLI or W@ Written in 68000 Assembler code @ Minimum 512k Amiga required @ NOT COPY PROTECTED

DigiSoft
12 Daroomer Street, Monto, Queensland, Australia 4810
Telephone (07) 277-3255

YES WE ACCEPT
Visa/Amex
Mastercard
Bankcard
All orders shipped promptly
All orders include AirMail

GILGANDRA ELECTRONICS

Our new store will be open soon. If you want help with your small business system . . . or service on your present system then give Gary Edwards a call to discuss it.

GILGANDRA ELECTRONICS
67 WRIGLEY STREET
GILGANDRA, NSW 2827
(068) 47 2491

A word in your ear . . .

If you are producing software, making hardware, or are an outlet for all things Amiga, then consider this . . .

Every copy of this magazine goes into the hands of a dedicated AMIGA user.

If you want them to know what you have to offer, then think about advertising your products with us.

We offer full typesetting facilities, and you will be surprised how economically you can tell your story.

INTERESTED? THEN RING TRANSACTOR SOON!
A Fast Flood Algorithm

Speed up your area fills by up to five times!

by Danny Ross

Danny Ross is an undergraduate in his final year of a B.Sc. Honours course in Computer Studies. His spare time is divided between developing routing algorithms for transputer networks, freelance Amiga programming and writing PC software. Living in a Gothic Lodge in South Wales, with two other 'code-junkies', his idea of relaxation is to explore every last Amiga library call.

Up until quite recently I was working on a conversion of a short PC program, written by a colleague in TURBO C, that was used for generating fractal landscapes from contour models. At that stage of development, the program used a mouse-driven system that allowed the user to draw in the map contours, from which it was to generate the landscape. The conversion was quite easy-going until it came to the part where my colleague had used TURBO C's flood fill routine - here the Amiga version seemed to just grind to a halt.

The problem was that the graphics.library routine Flood() was simply unable to match the TURBO C equivalent for speed, and things were beginning to look very bad indeed. Being a great believer in the graphics.library I found it very hard to accept that it was so slow; but a few experiments soon showed that the Flood() routine really was as lethargic as first impressions suggested and so I began to look for an alternative.

The following is a description of the Amiga's built-in Flood() function (with some guesses concerning its poor performance), a description of a replacement algorithm to perform the same function, and finally an implementation written in C and Assembler which outperforms the Flood() function by a factor of around 500%.

Flood()

As part of the graphics.library, Flood() performs the rarely-required function of filling an unknown shape with a set colour. Although it often takes a back seat to the AreaFill routines, Flood(), when required, is very useful. It needs only the (X,Y) coordinates of a point within the shape to be filled.

However, AreaFill is able to make use of the Blitter to perform its rendering, while Flood() is very CPU intensive. The result is, unfortunately, that AreaFill is blisteringly fast and Flood() is diabolically slow.

While the algorithm used for Flood() is good, the implementation has problems. If you have ever been pushed to use ReadPixel() or WritePixel() you may have noticed that they are awfully slow. The reason for this is that the complexity of the Amiga graphics structures introduces a very high overhead for single read and write operations. Both the ReadPixel() and WritePixel() functions are subject to this overhead and it appears that Flood(), using these functions, also suffers.

The line adjacency algorithm

If we are to write a replacement for Flood() we will need an area filling algorithm, and a with a reasonable performance at that. Such an algorithm is called line adjacency.

The basic strategy of the line adjacency fill is to locate each group of horizontally connected pixels in the interior of the region. It starts with a seed pixel known to be in the region's interior and scans left and right to find the start and end of the seed pixel's row. The entire row is then filled.

The algorithm proceeds by locating all groups of horizontally connected pixels that are vertically adjacent to the group it just scanned. Each time it finds an adjacent group of not-yet-filled pixels, LineAdjacencyFill() is called recursively to fill them. The algorithm terminates when all interior pixels have been filled.

The following describes the route taken by LineAdjacencyFill(). It is called initially with the seed pixel coordinates at R7 and an initial direction of down. The algorithm scans left and right to find the start and end positions of the first group, which is then filled using the graphics.library Draw() function, which is very much faster than WritePixel().

The next stage of the algorithm is to find and fill adjacent line segments in the same direction. As the first group was E6-B9, LineAdjacencyFill() will prepare to test all pixels in the line segment F6-P9. At P6, a check is made to see if the pixel is ripe for filling (and thereby part of a horizontal group). If it is, then LineAdjacencyFill() is called recursively to fill it.
Upon return from the call, the algorithm will continue along the F6-F9 line; however, the previous invocation of LineAdjacencyFill() will have performed a left-right scan on the F-row from F6 and so, as a bonus, we can step past the right limit in the safe knowledge that the skipped pixels have been tested already. In the example, the second call will have found a line segment of F2-F7 and filled it, so we can start again at F8.

Line adjacency is fully recursive and so the second invocation will continue after locating F2-F8 and will start to check the pixels from G2-G7. This will generate a third call (as G2 is a suitable candidate) and G2-G11 will be filled. Again, H2-H11 is tested but none of the pixels will result in a call to LineAdjacencyFill() as they are all border pixels and so the third invocation moves on to the next stage - a check in the opposite direction.

For optimal speed, the reverse check is only performed on those ranges that are outside of the parent's segment (the variables supplied as prevxl and prevxr in the program). In the example, the current leftmost pixel is G2 which is no further left than F2 from the line above and so nothing is done.

On the right hand side, a check is done from F8 to F11, resulting in a call to LineAdjacencyFill() at F9, filling the line F9-F11. This invocation has a short life; the first 'same direction' check finds no candidates (F9 is already filled) and the reverse check generates no worthwhile ranges. It returns, allowing invocation 3 to continue its reverse search at F12, which is outside its ranges and so it returns.

Invocation 2 continues (phew!) and abandons its 'same direction' check (it was only going to do G2-G7, but invocation 3 found and filled G2-G11) to start its reverse check. This finds E2 as a candidate and triggers a series of LineAdjacencyFill() calls that fill E2-E3, D2-D11, C2-C11, and B2-B11.

Eventually they all return to invocation 2 which, bypassing a pointless right-hand reverse direction check, returns to invocation 1. This starts its reverse direction checks on D6 and D8-D9 (the first invocation results in prevxl and prevxr equal to the seed x-coordinate, E7 in the example). As it happens, the whole of the D-row is already filled, and so the first invocation returns, the area fully traversed.

The advantages of LineAdjacencyFill() over some of the simpler fill algorithms are significant. Most immediate (for an Amiga implementation) is that pixel reads are kept to a minimum and filling is performed on a line basis. A C implementation of LineAdjacencyFill() follows the same fill pattern as the graphics library's own Flood() function on an almost one-to-one basis (in fact, so closely that it seems almost certain that at the heart of Flood() there sits a line adjacency algorithm waiting to break free).

But why bother with all that code if there is no actual improvement to be gained? Well, one of the reasons LineAdjacencyFill() manages its reasonable performance is its optimisation by using line filling - with a known start and end position we can perform very fast line drawing, much faster than repeatedly using WritePixel().

The next obvious area for optimisation is ReadPixel(), and whilst Line Adjacency reduces the number of calls to ReadPixel(), some parts of the algorithm (noticeably scanleft and scanright) still rely heavily on individual pixel reads. As we may well be performing several hundred reads each line, it makes sense to optimise the methods used to implement scanleft and scanright.

The streamRead() function

To provide a fast replacement for ReadPixel(), we can take advantage of the nature of scanleft and scanright. Knowing that they will require sequential, unidirectional reads at a constant y-position we can prepare an environment that, after initialisation, can perform very fast stream reads with none of the usual structure-access overheads. _initStreamRead and streamRead (listing 2) have been written in assembler and are the only reason for the five-fold performance increase from FastFlood().

The technique for performing streamRead is illustrated in figures 2, 3a and 3b.
of the screen’s bitplanes and the current pixel position within the word is calculated (Figure 2).

These words are then prepared for the stream read by shifting the pixels to be read to the edge. In the example in figure 2, the pixel offset has been calculated as 14. If we are reading left then pixels 14 and 15 will be required and if we are reading right then we will need pixels 14 to 0. In the event of an initialisation for a left scan (Figure 3a), the _initStreamRead() function shifts the word 14 pixels to the right, bringing bits 14 and 15 to the edge.

Once the initialisation has been performed, the path is clear for _streamRead() to move across the raster line very quickly. It should be noted that, as the colour index is built up by shifting bits into an accumulating register [160-161 and 200-201], the order in which the bitplanes are accessed is very important.

As _streamRead() progresses, it will shift these bits out of the word from each bitplane (once per pixel) to build a colour index. When the useful bits have been absorbed (two in this case), _streamRead() will go to memory to fetch the next consecutive word. On the same principle, an initialisation for a right scan (Figure 3b) will bring the bits to the left hand edge.

Within the BitMap structure there exists an array of up to 8 pointers, each one containing the address of a bitplane from the display. As the pointers are stored in low priority order, a normal traversal of the pointer array would result in an inverted colour index and so lines 70-74 are introduced to pick up the bitplane pointers in reverse.

Implementation

The FastFlood() function is contained within the fastflood.h include file (listing 1) - this isn't a perfect solution, but it makes things simpler. If you were using Lattice C and BLink you may like to make FastFlood() a self-contained function and PRELINK it together with the fastpixel.o binary file produced from listing 2.

To use FastFlood() you will need to assemble fastpixel.asm to produce the fastpixel.o file. Then, any program which requires the FastFlood() function must have the line:

```c
#include "fastflood.h"
```

at the start (see listing 3). The whole caboodle can then be compiled and linked with fastpixel.o and the standard C startup modules.

Limitations

Well - nothing's perfect, and unfortunately FastFlood() is no exception. There are a couple of quirks in the function that should be understood before it is used. Firstly, there is no check in _streamRead against the RastPort’s boundaries. I decided against this as, in most cases, it would be an unnecessary overhead. Having said that, if a flood is allowed to leak to an unbounded area then the scanleft() and scanright() functions will just hang a 7 and wander off through memory.

The second problem occurs with the need for the LockLayer() call. As we are scanning left and right, the user must not be able to move the window - the plane pointers will still address the same physical screen location, which may now be outside the area boundaries. LockLayer() will suspend any task attempting to alter the designated RastPort’s layer structures (such as moving the window or clicking on a gadget). If you try moving the window with Flood() you will notice the same effect. However, FastFlood() is very stack intensive due its recursive nature and without enough stack allocation (about 25k should do for the demonstration program) it may cause an overflow. If this happens in between LockLayer() and UnlockLayer() then we could be in big trouble.

The first limitation is easy to overcome as the window’s boundaries are available right at the beginning of FastFlood() where ff_xoff and ff_yoff are initialised from the RastPort’s Layer’s ClipRect’s Border structure. These could be passed to _initStreamRead() with a ‘current x-coordinate’ value maintained and tested by _streamRead().

The second limitation is rather harder to overcome and the only solution I could see would be to lock and unlock the layer structure at the beginning and end of each call to scanleft() and scanright(). These functions make no recursive calls and so are relatively safe from the stack overflow trap.

The demonstration program

Should you decide to enter and run the demonstration program in listing 3, you will be presented with a bland but serviceable screen endowed with several gadgets and a drawing area.
pressing the left mouse button in the drawing box you may sketch areas to test out the fill algorithm. You can select which algorithm the program will use (Flood() or FastFlood()) by clicking in either one of the top two gadgets.

Further down the screen there are two more gadgets to increase or decrease the current fill colour, and still further down, two gadgets to reset (clear) the drawing area and quit the program.

Filling is performed by pressing the right mouse button anywhere within the drawing area but be careful there are no holes in your sketch or the flood may leak out and fill the whole area.

Conclusion

FastFlood() combined with the fastpixel.asm functions provides an effective replacement for the Flood() function, with greatly improved performance. For still further improvements, the entire algorithm could be written in assembler (and indeed has been) but for the purposes of this article I decided against it. The main reason was that as stand-alone functions, _init-StreamRead() and _streamRead() may be very useful in other programs that require pixel reads in groups of more than about 4 pixels.

The principle behind the _stream routines is one that is applied frequently in optimising areas of code - identify that code which is executed most often and attempt to perform as much preparation as possible before entering the repeating sections. In this way, any removal of redundant computation from within the loop will have the greatest effect on performance.

Acknowledgements


Listing 1: Fastflood.h

```c
/*
 fastflood.h

 Fast Flood Fill - FastFlood() routine, by Danny Ross 13th January 1989
 A product of the Johnny Appleseed Software Development Co.
 Requires the assembler support routines from fastpixel.s
 */

#include <exec/types.h>
#include <graphics/textport.h>
#include <graphics/clip.h>
#include <graphics/gl.h>
#include <stdio.h>

extern void initStreamRead();
extern int streamRead();

SHORT LineAdjacencyFill(x, y, d, prevx, prevy);
void scanleft(int, scanright(int);

struct LayerBase
struct Layer

SHORT xoff, yoff;

void FastFlood(xoff, yoff, x, y, d)
struct RasterPort *rp;
short x, y, d;

if (LayerBase->LayerLibrary->OpenLibrary("layers.library", 0))
    LockLayer(Layer->LayerLibrary->layer);
    ff_rp = rp;

    if (mode == 0)
        ff_outPen = (int)(ff_rp->M1Pen); else ff_outPen = ReadPixel(rp, xp, yp);
    ff_inPen = (int)(ff_rp->M1Pen);

    LineAdjacencyFill((SHORT)x, (SHORT)y, (SHORT)x, (SHORT)y);
    UnlockLibrary(LayerBase);

    layer = LayerBase->Layer;
    cl-layer->ClipRect;
    xoff = Hi((cl-layer->Bounds)->MaxX);
    yoff = Hi((cl-layer->Bounds)->MaxY);

    for (x = xoff; x <= x; x++)
        for (y = yoff; y <= yoff; y++)
            DrawLayer(x, y, d, x, y, d);
```
void scanleft(x,y)
SHORT *x, *y;
{
  register int v;

  initStreamRead((int)ff rp, (int)(ff.yoff+(ff.yoff+y), (int)-1),
                  (int)(ff.xoff+y), (int)-1);
  do
    { 
      --(*x); v=streamRead();
    } while ((v!=ff_outPen) && (v!=ff_inkPen));
  ++(*x);
}

Listing 2: fastflood.asm

HiSoft GenAm 680x0 Macro Assembler v2.08

1 00.00000000
2 00.00000000
3 00.00000000
4 00.00000000
5 00.00000000
6 00.00000000
7 00.00000000
8 00.00000000
9 00.00000000
10 00.00000000
11 00.00000000
12 00.00000000
13 00.00000000
14 00.00000000
15 00.00000000
16 00.00000000
17 00.00000000
18 00.00000000
19 00.00000000
20 00.00000000
21 00.00000000
22 00.00000000
23 00.00000000
24 00.00000000
25 00.00000000
26 00.00000000
27 00.00000000
28 00.00000000
29 00.00000000
30 00.00000000
31 00.00000000
32 00.00000000
33 00.00000000
34 00.00000000
35 00.00000000

; produce linkable code.

; Assembler support routines for FastFlood() function
; A product of the Johnny Appleseed Software Development Co.
; Written by Danny Ross, 19th January 1989

; C Definitions -

; void initStreamRead(rp, xp, yp, direction)
; struct RastPort *rp;
; int xp, yp;
; int direction;
;
; C Example call -

; InitStreamRead(rp, xp-1, yp, -1); /* read left */
; do
  xp--; v=streamRead();
while (v!=border_colour);

; InitStreamRead(rp, xp+1, yp, 1); /* read right */
; do
  ++(*x); v=streamRead();
while (v!=border_colour);

; InitStreamRead(rp, xp, yp+1, 1); /* read down */
; do
  ++(*y); v=streamRead();
while (v!=border_colour);

; InitStreamRead(rp, xp, yp-1, -1); /* read up */
; do
  --(*y); v=streamRead();
while (v!=border_colour);
; stack offset to reach past saved
; registers

; A0->RastPort
; A0->BitMap
; D1=BytesPerRow
; D0=xpos
take a copy of this x

; D0=4 of bitplanes
; store this for later

; D0=4•(1 of bitplanes)
; A0->end of plane ptrs. This is
done so we can go thru the ptrs
; in reverse (see text)
; D0 is back to normal

; if it's +ve we don't do anything
else reverse 0..15->15..0

; store data word in table
; repeat for each bitplane
tst.w D3 ; check direction again
bpl.a bitRht ; we are heading right...
mov.w #68DDDD,DD ; prepare bit marker
tst.w D4 ; do we have to shift it ?
beq.s initFin ; no
lsr.w D4,D0 ; correct for mid-word data
bra.s initFin

; prepare bit marker
; do we have to shift it ?
; no
; correct for mid-word data
; store bit marker

D6=4 of bitplanes (-1 for DBF)
record the bitmap depth
record the scan direction
restore registers
return to C

The Initialisation routine is finished.

The streamRead() function returns the value of the current pixel and
then moves on to the next, reloading data words if a word boundry is
reached.

XDEF _streamRead

; The next section of code reads the colour at the current pixel
; and checks to see if we have hit a border. If not we move on to the
; next pixel, reloading the data tables if necessary.

; take out bit
; and shift it into colour
again for the next plane

lea.1 storeW(PC),AO
moveq 40,00
; save A3
; initialise D1 as 4 of planes
; initialise AO as -> to data
; set colour to 0

; the next section of code reads the colour at the current pixel
; and checks to see if we have hit a border. If not we move on to the
; next pixel, reloading the data tables if necessary.
In the Transactor, the AMIGA

170 00.000000C4  resetR move.l (A0),A3 ; A3 -> current address in plane
171 00.000000C4 2650 move.w 1450001, (Al) ; reset bit marker
172 00.000000C5 D7FC00000002 addw #2, A3 ; move on to next word
173 00.000000CC 32D3 move.w (A3), (A1)+ ; read next word and store in table
174 00.000000CC 20CB move.l A3, (A0)+ ; store bitplane address
175 00.000000DD 51C9FFFF2 dbf.w D1, resetR ; and repeat for all planes
176 00.000000BD 328C0001 move.w #$0001, (Al) ; reset bit marker
177 00.000000BB 265F skipRR move.l (A7)+,A3 ; restore A3
178 00.00000075 rts
179 00.000000CC ; Fast Read Left
180 00.000000DC ; The next section of code reads the colour at the current pixel
181 00.000000DE ; and checks to see if we have hit a border. If not we move on to the
182 00.000000DE ; next pixel, reloading the data tables if necessary.
183 00.000000DE ;
184 00.000000DE

185 00.000000DD 2F0B readL move.l A3, -(A7) ; save A3
186 00.000000DE

187 00.000000DE 32A00D64 move.w storeDep(PC), D1 ; initialise D1 as # of planes
188 00.000000DE 41FA004E lea.l storeW(PC), A0 ; Initialise A0 as -> to data words
189 00.000000DE 7000 moveq #0, D0 ; set colour to 0
190 00.000000DE

191 00.000000DE 2650 resetL move.l (A0), A3 ; A3 -> current address in plane
192 00.000000DE 32E3 move.w -(A3), (A1)+ ; read next word and store in table
193 00.000000DE 265C 00C0 movw A3, (A0)+ ; store bitplane address
194 00.000000DE 51C9FFFF2 dbf.w D1, resetL ; and repeat for all planes
195 00.000000DE 328C8000 move.w t$8000, (A1) ; reset bit marker
196 00.000000DE 265F skipLR move.l (A7)+, A3 ; restore A3
197 00.000000DE rts
198 00.000000DE

199 00.000000DE 6408 leftPL roxr.w (A0)+ ; take out bit
200 00.000000DE 3350 roxl.w #1, D0 ; and shift it into colour
201 00.000000DE 51C9FFFFA dbf.w D1, leftPL ; again for the next plane
202 00.000000DE 661A bne.s skipLR ; not zero so don't reload
203 00.000000DE 41FA001C lea.l storeP(PC), A0 ; prepare to get the next word
204 00.000000DE E2DD lea.l storeP(PC), A0 ; ... of bitplane data
205 00.000000DE 661A bne.s skipLR ; not zero so don't reload
206 00.000000DE

207 00.000000DE 41FA001C lea.l storeP(PC), A0 ; prepare to get the next word
208 00.000000DE 43FA003B lea.l storeW(PC), A0 ; ... of bitplane data
209 00.000000DE 32A00D64 move.w storeDep(PC), D1 ; D1 = # of bitplanes
210 00.000000DE 2650 resetL move.l (A0), A3 ; A3 -> current address in plane
211 00.000000DE 32E3 move.w +(A3), (A1)+ ; read next word and store in table
212 00.000000DE 00C0 movw A3, (A0)+ ; store bitplane address
213 00.000000DE 51C9FFFF2 dbf.w D1, resetL ; and repeat for all planes
214 00.000000DE 328C8000 move.w #$8000, (A1) ; reset bit marker
215 00.000000DE 265F skipLR move.l (A7)+, A3 ; restore A3
216 00.000000DE rts

217 00.000000DE ; Working data initialised by initStreamRead().
218 00.000000DE ;
219 00.00000100 4675
220 00.00000100 4008 ; plane pointers
221 00.00000100 4008 ; current data words
222 00.00000100 4008 ; bit marker (possibly)
223 00.00000100 4008 ; depth of bitmap
224 00.00000100 4008 ; direction
225 00.00000100 4008 end
Listing 3: demo.c

/ *

Fast Fill routine - Demonstration program by Danny Ross.
A product of the Johnny Appleseed Software Development Co.
Written using CygnusED Professional, Compiled with Lattice C version 4.0
Requires fastflood.h

*/
#include <stdio.h>
#include <intuition/intuition.h>
#include <intuition/intuitionbase.h>
#include <graphics/gfx.h>
#include <graphics/gfxmacros.h>
#include "dfl;fastflood.h"

struct TextAttr Myfont =
{
    "topaz.font",
    TOPAZ EIGHTY,
    FS NORMAL,
    PFP_ROMFON,
    };

#define gwi 120
#define ghi 20
#define FLOOD 0
#define FASTFLOOD 1

SHORT Outline11] = { 0,2,2,0,119,0,121,2,121,17,119,19,2,19,0,17,2,19,17,2,19,0,17,2,19,0,17,2,19,0,17,2,19,0,17,2,19,0,17,2,19,0,17,2,19,0,17,2,19,0,17,2,19,0,17,2,19,0,17,2,19,0,17,2,19,0,17,2,19,0,17,2,19,0,17,2,19,0,17,2,19,0,17,2,19,0,17,2,19,0,17,2,19,0,17,2,19,0,17,2,19,0,17,2,19,0,17,2,19,0,17,2,19,0,17,2,19,0,17,2,19,0,17,2,19,0,17,2,19,0,17,2,

struct IntuiText flood_text = { 15,1,JAM1,5,6,4MyFont," Flood",NULL };
struct IntuiText fastflood_text = { 15,1,JAM1,5,6,4MyFont," Fast Flood",NULL };
struct IntuiText colup_text = { 15,1,JAM1,5,6,4MyFont," 4",NULL };
struct IntuiText coldn_text = { 15,1,JAM1,5,6,4MyFont,' 1',NULL };
struct IntuiText reset_text = { 15,1,JAM1,5,6,4MyFont," Reset",NULL };
struct IntuiText quit_text = { 15,1,JAM1,5,6,4MyFont," Quit",NULL };

struct Border gadget_border2 = { 0,0,15,0,JAM1,9,4Outline2[0],NULL };
struct Border gadget_border = { 0,0,8,0,JAM1,9,4Outline1[0],&gadget_border2 };

struct Gadget flood_gadget =
{  
    NULL,435,40,gwi,ghi,GADGCMP,RELVERIFY,
    BOOLGADGET, (APTR4gadget_border,NULL, &flood_text, 0x0000, NULL, 0, NULL

};

struct Gadget fastflood_gadget = 
{  
    flood_gadget,435,70,gwi,ghi,GADGCMP,RELVERIFY,
    BOOLGADGET, (APTR4gadget_border,NULL, &fastflood_text, 0x0000, NULL, 1, NULL

};

struct Gadget colup_gadget =
{  
    fastflood_gadget,370,120,gwi,ghi,GADGCMP,RELVERIFY,
    BOOLGADGET, (APTR4gadget_border,NULL, &colup_text, 0x0000, NULL, 2, NULL

};
struct Gadget coldn_gadget =
{
    &coldn_gadget, 500, 120, gwi, GADGNCOMP, RELVERIFY,
    BOOLGADGET, (APTR) &gadget_border, NULL, &coldn_text, 0x0000, NULL, 3, NULL
};

struct Gadget reset_gadget =
{
    &reset_gadget, 370, 150, gwi, GADGNCOMP, RELVERIFY,
    BOOLGADGET, (APTR) &gadget_border, NULL, &reset_text, 0x0000, NULL, 4, NULL
};

struct Gadget quit_gadget =
{
    &reset_gadget, 500, 150, gwi, GADGNCOMP, RELVERIFY,
    BOOLGADGET, (APTR) &gadget_border, NULL, &quit_text, 0x0000, NULL, 5, NULL
};

struct NewScreen ns =
{
    0, 0, 640, 250, 4, 0, HIRES, CUSTOMSCREEN, MyFont,
    "Flood Fill Demonstration Screen", NULL, NULL
};

struct NewWindow nw =
{
    0, 11, 640, 239, 0, 1, CLOSINGO, GADGETUP, MOUSEBUTTONS,
    WNDOWNCLOSE, WNDOWNDRAG, WCAREKEPRESHT, WMTRAP|ACTIVATE, &quit_gadget, NULL,
    "By Danny Ross, 10th March 1989", NULL, NULL, 0, 0, 0, 0, CUSTOMSCREEN
};

struct RastPort *rp;
struct Screen *scrn = NULL;
struct Window *wndo = NULL;
struct ViewPort *vp;

int bpen, ipen, col = 1;
int fillStyle = FLOOD;

struct IntuitionBase *IntuitionBase = NULL;
struct GfxBase *GfxBase = NULL;

void initialise(), abEnd(), setfillstyle();

void initialise()
{
    if ((IntuitionBase=(struct IntuitionBase*)OpenLibrary("Intuition.library", 0))==NULL) abEnd("Can't open Intuition Library");
    if ((GfxBase=(struct GfxBase*)OpenLibrary("Graphics.library", 0))==NULL) abEnd("Can't open Graphics Library");
    if ((scrn=(struct Screen *)OpenScreen(6ns))==NULL) abEnd("Can't open Screen");
    nw.Screen=scrn;
    if ((wndo=(struct Window *)Openwindow(&nw))==NULL) abEnd("Can't open Window");
    vp=(struct ViewPort *)ViewPortAddress(wndo);
    rp=wndo->RPort;
    SetSoftStyle(rp, FSF_BOLD, 95);
    SetAPen(rp, 15); RectFill(rp, 20, 30, 320, 230);
    SetAPen(rp, 0); RectFill(rp, 21, 31, 319, 229);
    SetAPen(rp, col); Move(rp, 445, 109); Text(rp, "Fill Colour", 11);
    SetAPen(rp, 15); bpen=15; SetOPen Irp, 15); setfillstyle(FLOOD);
}
void abEnd(s)
    char *s;
    {
        if (s) printf("%s\n", s);
        if (scrn) CloseScreen(scrn);
        if (wndo) CloseWindow(wndo);
        if (GfxBase) CloseLibrary(GfxBase);
        if (IntuitionBase) CloseLibrary(IntuitionBase);
        exit();
    }

void floodfill(fillStyle, col, xp, yp)
    int col, xp, yp;
    {
        SetAPen(rp, col); ipeo-ccol;
        switch (fillStyle)
            {
            case FLOOD : Flood(rp, 0, xp, yp); break;
            case FASTFLOOD : FastFlood(rp, 0, xp, yp); break;
            default : break;
            }
        SetAPen(rp, 151);
    }

main()
    {struct IntuiMessage *message;

        int mx, my, quit, lost;
        ULONG class;
        USHORT code;

        struct Gadget *gadget;

        initialise();

        quit=FALSE;
        while (quit==FALSE)
            {
                Wait(1<< (wndo->UserPort->mp SigBit));
                while ((message=( struct IntuiMessage *)GetMsg wndo->UserPort))!=NULL)
                    {
                        class=message->Class; code=message->Code;
                        mx=message->MouseX; my=message->MouseY;
                        gadget=( struct Gadget *)message->TAddress;
                        ReplyMsg(message);
                        switch (class)
                            {
                            case CLOSEWINDOW :
                                quit=TRUE;
                                break;
                            case GADGETUP :
                                switch (gadget->GadgeID)
                                    {
                                    case 0 : { Setfillstyle(FLOOD); break; }
                                    case 1 : { Setfillstyle(FASTFLOOD); break; }
                                    case 2 :
                                        { col={(col+1)%15}; SetAPen(rp, col);
                                            Move(rp, 445, 109); Text(rp, "Fill Colour", 11); SetAPen(rp, 151);
                                            break;
                                        }
case 3:
{
    if (--col<0) col=14;
    SetPen(rp,col);
    Move(rp,445,109); Text(rp,"Fill Colour",11); SetPen(rp,15); break;
}
case 4:
{
    SetPen(rp,15); RectFill(rp,20,30,320,230);
    SetPen(rp,0); RectFill(rp,21,31,319,229);
    SetPen(rp,15);
    break;
}
case 5: { quit=TRUE; break; }
break;
case MOUSEBUTTONS:
{
    if (code==SELECTDOWN)
    {
        lost=TRUE;
        while (GetMsg(wnd->UserPort)==NULL)
        {
            mx=wnd->MouseX; my=wnd->MouseY;
            if ((mx>=20) && (mx<320) && (my>=30) && (my<230))
            {
                if (lost) lost=FALSE; Move(rp,mx,my);
                Draw(rp,mx,my);
            }
            else
            {
                lost=TRUE;
            }
            SendMsg(message);
        }
        else if (code==MENU_DOWN)
        {
            mx=wnd->MouseX; my=wnd->MouseY;
            if ((mx>20) && (mx<320) && (my>=30) && (my<230))
            {
                floodfill(fillStyle,col,mx,my);
            }
            break;
        }
        default:
        {
            break;
        }
    }
}
abEnd(NULL);

void setfillstyle(style)
{
    int style;
    Move(rp,380,29);
    fillStyle=style;
    switch(fillStyle)
    {
    case FLOOD: { Text(rp,"Flood Algorithm - Flood()",29); break; }
    case FASTFLOOD: { Text(rp,"Flood Algorithm - FastFlood()",29); break; }
    }
}
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 “|” 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 | program2 -y | 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)". "MOVEL #$3EE,D0" becomes "MOVEL #$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)