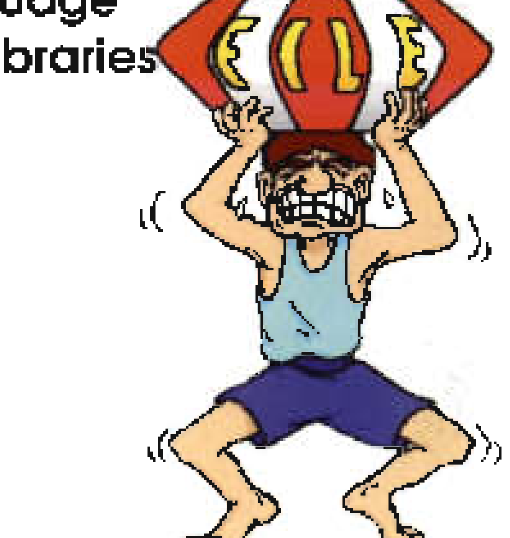
Building an Audio Digitizer

# A Look at Compression

- True F-BASIC
- A Date with TrueBASIC
- A Better Way to C
- Huge Numbers Part 2
- Programming the Amiga in Assembly Language
- AmigaDOS Shared Libraries




bizcomputers.cz

# CATCH THIS.

Introducing FreshFish™, a unique CD-ROM series that provides the Amiga community with hundreds of megabytes of the very latest in freely redistributable software.

The FreshFish CD ROM series is produced directly by Fred Fish, who has been working to supply Amiga users with high-quality, freely redistributable software since the Amiga's introduction in 1985. FreshFish CDs, published every 6 to 8 weeks, contain over 100 Mb of newly submitted material in both BBS ready (archived) and ready-to-run (unarchived) form. Also included are over 200 Mb of ready-to-run GNU software (EMACS, C/C++ compiler, text processing utilities, etc.) with full source code included, and up to 300 Mb of other useful utilities, games, libraries, documentation and hardware/software reviews.

Two compilation CDs will also be available. The FrozenFish™ series will be published every 4 to 6 months as a compilation of the most recent material from the FreshFish CDs.

GoldFish™, a two disc CD ROM set, will be available in April 1994. This set will contain the entire 1,000 floppy disk "Fred Fish" library in both BBS ready and unarchived form!

FreshFish, FrozenFish, and GoldFish may be purchased by cash, check (US dollars), Visa, or MasterCard, from Amiga Library Services for $19.95 each (plus $3 shipping & handling in the U.S., Canada or Mexico, $5 elsewhere).

Fax or mail orders and inquiries to:

**Amiga Library Services**
**610 North Alma School Road, Suite 18**
**Chandler, AZ 85224-3687 USA**
**FAX: (602) 917-0917**

DJComputers.cz

AC's TECH / AMIGA

ADMINISTRATION

Publisher: Joyce Hicks
Assistant Publisher: Robert J. Hicks
Administrative Asst.: Donna Viveiros
Circulation Manager: Doris Gamble
Asst. Circulation: Traci Desmarais
Traffic Manager: Robert Gamble
Marketing Manager: Ernest P. Viveiros Sr.

EDITORIAL

Managing Editor: Don Hicks
Editor: Jeffrey Gamble
Hardware Editor: Ernest P. Viveiros Sr.
Video Consultant: Oran Sands
Illustrator: Brian Fox

ADVERTISING SALES

Advertising Manager: Traci Desmarais

1-508-678-4200
1-800-345-3360
FAX 1-508-675-6002

AC's TECH For The Commodore Amiga™ (ISSN 1053-7929) is published quarterly by PiM Publications, Inc., One Currant Road, P.O. Box 2140, Fall River, MA 02722-2140.

Subscriptions in the U.S., 4 issues for $44.95; in Canada & Mexico surface, $52.95; foreign surface for $56.95.

Application to mail at Second-Class postage rates pending at Fall River, MA 02722.

POSTMASTER: Send address changes to PiM Publications Inc., P.O. Box 2140, Fall River, MA 02722-2140. Printed in the U.S.A.
Copyright© 1994 by PiM Publications, Inc. All rights reserved.

First Class or Air Mail rates available upon request. PiM Publications, Inc. maintains the right to refuse any advertising.

PiM Publications Inc. is not obligated to return unsolicited materials. All requested returns must be received with a Self Addressed Stamped Mailer.

Send article submissions in both manuscript and disk format with your name, address, telephone, and Social Security Number on each to the Editor. Requests for Author's Guides should be directed to the address listed above.

AMIGA™ is a registered trademark of Commodore-Amiga, Inc.

# Contents

Volume 4 Number 2

3 **True F-BASIC**
*by Roy M. Nuzzo*

9 **A Look at Compression**
*by Dan Weiss*

14 **A Date with TrueBASIC**
*by T. Darrel Westbrook*

22 **Building an Audio Digitizer**
*by John Iovine*

26 **A Better Way to C**
*by Paul Gittings*

42 **Huge Numbers Part 2**
*by Michael Greibling*

54 **Programming the Amiga in Assembly Language: Using the Math Co-processor**
*by William P. Nee*

68 **AmigaDOS Shared Libraries**
*by Daniel Stenberg*

## Departments

2 Editorial
40 List of Advertisers
41 Source and Executables ON DISK!

# Startup Sequence

## Hug A Beginner

AC's TECH is devoted to the Amiga user who wants to go deeper into the Amiga. Through hardware projects like *Building an Audio Digitizer* (on page 22 of this issue), AC's TECH readers have been able to understand more about the hardware capabilities of the Amiga and how the Amiga relates to other peripheral hardware. In addition, software has also been a major part of AC's TECH as we continue to offer programs in True BASIC, C, Assembly Language, and more. These articles and programs are a catalyst to users who have an understanding of the Amiga, but want examples to help them create their own works.

### Teaching To Learn

A very important facet of teaching anything is that once we focus on how to explain something to another person, we create a clearer picture of our subject for ourselves. We begin examining the project or subject from a variety of different views to be able to explain the subject in various ways so it is more readily understood by the student. This examination often opens barriers to our own understanding.

If we have become complacent with an idea or a concept, we begin to view the idea from a static angle. However, if we are challenged to look beyond (and around) mental blocks, we begin to see the concept anew. It is as if we have found a door in our house that leads to a completely new living area.

In the movie "Dead Poets Society," Robin Williams plays a teacher in a rural male school. At one point, he challenges his students to stand on his desk and look at the classroom from a different perspective. The students discover a view of their world that they never knew existed. What they take with them is the ability to challenge the normal and consider different concepts.

Teaching informs the teacher. By becoming secure in the details and concepts, the teacher becomes more confident in the subject. This makes better teachers (as long as they remain open to different views) and it makes better hardware enthusiasts.

**Each of us can point to at least one person who helped us in the past become what we are today.**

None of the products and articles discussed in AC's TECH should be taken at face value. Each article is a means to an end. The staff at AC remain dedicated to creating more informed Amiga users. *Amazing Computing For The Commodore Amiga* consistently offers product reviews, new product announcements, bug fixes, hints and tips, and more. *AC's Guide To The Commodore Amiga* maintains a database of software and hardware tools available to the Amiga market.

The aim of AC's TECH is not to make everyone a hardware hacker or software guru although we would be very excited if it did. The purpose behind AC's TECH is to make users more comfortable with the thought of creating a software program or combining hardware peripherals on the Amiga. If the ideas and projects presented here help to create superior hardware developers or software designers, then we are more than pleased.

This means that while the projects and articles presented in AC's TECH require technical knowledge and should not be attempted unless the Amiga user is aware of the possible consequences, beginners should not be dissuaded from getting a more experienced user to help them. This "buddy system" will not only create more competent users for the Amiga, but it will also challenge the more experienced user.

### We All Started Empty

Fortunately none of us started life knowing everything. We have all needed to learn things through time. Each of us can point to at least one person who helped us in the past become what we are today. The fortunate of us can point to a number of people who have made a difference when they took the time to help us understand what they already knew.

There are also those individuals who can point with pride to an accomplishment of someone they have helped and know that it was their effort that prepared the individual for the achievement. This is not egotistical or ungenerous. These people are genuinely proud of what their friend or student has been able to do. Their pride in being part of that is very human.

### Asking More

I have often asked our readers to be more involved in the Amiga market. I have asked them to talk with their dealers and with developers to offer positive suggestions on how products or services may be improved. I feel that anyone who has an investment in the Amiga, either financially or mentally, owe it to themselves to do what they can to create an Amiga marketplace that services them. What we can not forget is that new users need our input even more.

When a newly purchased Amiga (new or used) is sitting in front of its new owner, immediately there are a thousand questions to be answered. While it is important that users should learn as much as they can by themselves, it is also important that they have a source for experienced help.

A strong argument could be made that this is what the dealer is paid to do. However, a dealer cannot be available whenever the new user stumbles over another new concept. While most dealers strongly support their customers, none of them are available as often as a new [illegible] crisis may occur.

Many new users are buying used machines from Amiga users who are trading up to more powerful Amigas such as the Amiga 4000 or 1200. These "new" machines require as much support as Amigas delivered from a showroom. While most users want to support someone who has purchased their old Amiga, there is always the call of the new Amiga to pull them away. Sometimes it is difficult to patiently explain a kickstart incompatibility when all you really want to do is fire up the latest and greatest graphics package on your fully expanded Amiga.

However, the student can surpass the teacher. The person who is just starting today can easily become your best resource in the future. Doesn't it make a lot of sense to help them now? You could need them.

Sincerely,

Don Hicks
Managing Editor



DJComputers.cz

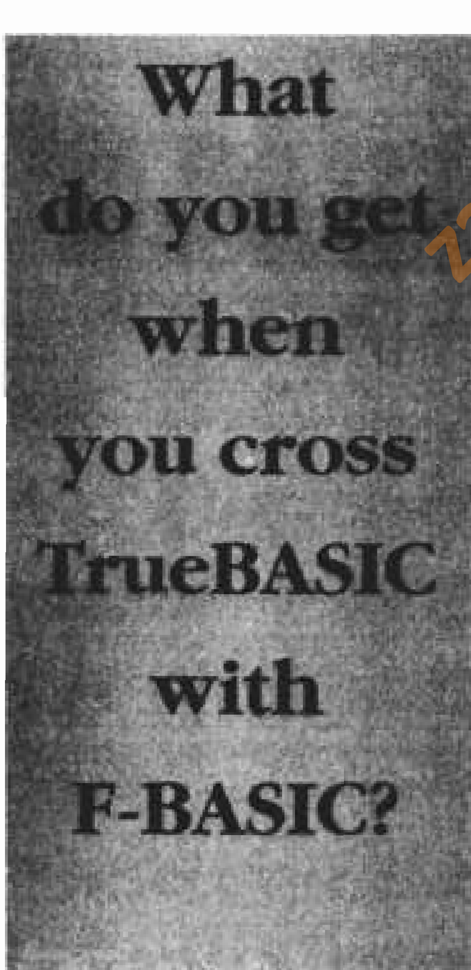
# True F-BASIC?

by Roy M. Nuzzo

Do you like weird? This is weird. However, it is good weird. I will show you how to write, debug, and run F-BASIC from True BASIC.

What?

No. Really. There is method in this madness. First the reasons why. True BASIC is very intuitive and easy to dream in. It is a super fast idea testing environment. The reasons are several. The language is very coherent and unfettered. The language editor is just simply the best thing out there for any file editing purpose. The editor takes macros to an unprecidented level. It can even nibble and edit its own structure. That is what this article will show you. True BASIC has some flaws. Overscan and the new AGA modes are not supported. There is no stand alone linker for the 32 bit machines.




F-BASIC has a single author feel, lots of idiosyncracies, very arcane command syntax structures, and mixed model borrowing from many languages (I feel at home with the throw back to Fortran66-like string handling, I grew up on that). However, what F-BASIC does, it does very very well. It does overscan and AGA graphics. It does many neat things directly that other languages require you to do with nasty add on support. It also drives you crazy with the most annoying text editor on Earth.

Solution. Take the True BASIC editor and turn it into an F-BASIC editor that writes, compiles, runs, and debugs F-BASIC.

The 'TBE' (we'll call the True BASIC editor 'TBE') has a 'Do' feature. You write what is essentially a program except that it is placed at the top of an external library as a first subroutine in that library. This library is simply a sequence of subroutines saved as a separate file (compiled or uncompiled). By issuing a command to 'Do filename', the first sub in that file is activated as if it were a stand alone program and executes as would a program.

However, it operates in the editor's background. While it is running, you are looking at the text in the editor. The 'Do Program' does not replace or cover the current file that you are looking at. The 'Do routine' may be passed a string argument. If you omit the string argument, it supplies its own as a null. That string might be a complex string of additional arguments but we do not need that here.

The 'Do program' automatically has access to the code (text) currently residing in the editor, and can read it and modify it right as you are looking at it. It can also do anything that any program can do such as send out system shell commands, organize and read files, whatever.

The TBE can also load binary files and edit directly by typing from the keyboard or by way of clever 'Do' programs. It can load a copy of itself and edit its own code and resave the modified editor in working tip top shape.

⇒

When the TBE is first started up, it reads a startup file called 'TBStartup' residing in the drawer called 'TBLib' located in the main TrueBASIC directory (which I have assigned as 'TB:'). This TBStartup file lists 'ALIAS' names for paths to various user libraries so that applications on different machines (even MAC or IBM) can use the same program path references. An alias (TBLibrary) may actually be 'Work:Lang/TB/TBLibrary' on your machine. It can also preload any resources (libraries) and key files as you wish. Any sub in a preloaded library is immediately available to typed commands in the command window.

You can also edit the help files which pop up at your desire. I never really needed them with True BASIC, but they sure are handy for the oddball commands of F-BASIC.

Also in the TB drawer is a subdirectory called 'TBLibrary' containing various support libraries including the one we will use named 'AmigaLib*'. This compiled library contains the subroutine named 'CLI()' which allows user programs to issue any shell command at runtime.

Example: Call CLI("copy RAM:# to df0: ALL") .

Following is a library containing only a 'Do Program'. You see it as a subroutine sitting at the top of the file. The 'EXTERNAL' statement merely tells the compiler to compile this file without looking for a main program (that makes this a library).

If you compile and then save the compiled library file as 'Foo', the command to activate the first sub as a 'Do Program' would merely be 'do foo'. The sub name is not used. This allows any library to have a housekeeping sub at the top to be activated without remembering that subs individual name.

I have the working drawer for F-BASIC assigned as 'FB:' on my computer. Also note that the code '...' for line continuation in this listing is not real, but there simply to allow magazine 4-column reading of the longer single line.

```
              !--------------------------
! 'FB.Do'          F-BASIC Compile Driver
!                    by Roy M. Nuzzo
!
! Compile and save as 'FB' in the 'TBDo' drawer.
! This is a 'do program' activated from within the
! True BASIC editor. Editor started from the 'FB:' path.
! Save the program text file in the 'FB:' directory.
! At the command window type "do FB, filename"
!     and press then enter.
! F-BASIC is case sensitive as are certain names.
! The True BASIC editor will launch FB.FB with the
! debugger option.
!
! If the file has an error, the debug option leaves a text
! file read by this 'Do' routine. The 'Do routine' returns
! to the editor with the line num. of the text listed
!
! At the editor command window type "do FB, filename"
! and you are placed on the offending line.
 --------------------------
          EXTERNAL
          sub FBcompile(lines$(), filename$)
   !                [illegible]
   !        All 'do' subs have these two arguments.
   !        The array is passed by the editor, itself.
   !        This array is the code actually residing
   !        in the editor.
              library "TBLibrary/AmigaLib*"
   !       this lib has the CLI() sub in it.
              let filename$ = trim$(filename$)
      ! the argument is the 'do FB, file'.
      ! [illegible]

          if len(filename$) = 0 then

   !----------------------
   ! No file name was supplied with do command.
   ! Try the file name currently in the editor.
   !
   !----------------------

   ask name filename$
           WHEN ERROR IN

     ! see if it is out there
    open #100: name "FB:" & filename$, organization...
                              ...TEXT, create old
     close #100

   USE
     ! If here, then save the current file name and run
     ! [illegible]. Return error code to editor & return
     ! control to editor. User needs to save file
     ! to FB:, then compile it.

     close #100

     cause error 1, "Save to FB: directory"
     ! The above words will appear
     ! in the editor message bar

   END WHEN

 END IF
          WHEN ERROR IN

     .                                  ! Remove old 'name.ERR' file, if
there,
     ! but don't crash if not.
     !--------------
      when error in
       call CLI("delete FB:" & filename$ & ".ERR")
      use
      end when
                   !
     ! RUN LAUNCH THE F-BASIC COMPILER (called 'FB')
     ! [illegible]
     ! [illegible]
     !-----------
     call CLI("FB " & filename$ & " opt-d")

     sound 1000,.01
                   ! [illegible] line counter
     let ErrFlag$ = "No Err."
     let count = 0
                WHEN ERROR IN
       !-------------
       ! Examine the resultant '.ERR' file.
       ! [illegible]
       ! Number of lines in that file = the line
       ! at which compile crashed.
       ! Count the lines.
       !-------------
       open #101: name "FB:" & filename$ & ".ERR", ...
                    ... organization TEXT, create old
                set #101: POINTER BEGIN

       do while more #101
          line input #101: x$
          let count = count +1
       loop

       let ErrStr$ = "Err at " & str$(count) & ...
                      ... [illegible]

       ! [illegible] will appear in the
       ! editor message bar.

       sound 2000,.01
              [illegible]

     END WHEN

     close #101

     ! Run the successfully compiled file.
     !
     call CLI("FB:" & filename$ & [illegible])
```



DJComputers.cz

```
[illegible]
    cause error 1, [illegible]
    exit sub
  end WHEN
  cause error 1, [illegible]
End SUB
```

What does this do? The editor launches the 'do' routine which evokes the F-BASIC compiler (called 'FB') with the option to save debug files as well. The absence of an ".EXT" file means that it compiled properly. The 'do' routine then issues the Shell command to run the compiled version of the program (name ends in ".bin").

When the running of the compiled program is completed, the True BASIC editor takes over again. The message bar in the editor echoes any messages passed to it by way of the 'cause error' commands.

By compiling this library and naming it "FB" and saving it to the TB:TBDo drawer, you can compile and run an F-BASIC program by typing the following in the command window of the editor:

```
[illegible]
```

That's it. But that's not enough. We want weird and we shall have it!

Clear the True BASIC editor. Load a new file. That file is the editor itself. Go ahead. It will load just fine.

Use the 'Find' function to find the string of characters "FORMAT" buried in the sea of binary characters. You will find "FORMAT" twice. Each time it is part of a text sequence "Do FORMAT" surrounded by gibberish. Keeping the same number of characters (6), delete "FORMAT" and type "FBASIC". Again this is required twice. The first occurance was the text seen in the editor pull down menu. The second time is the text actually issued to the outer Amiga shell by the editor when the corresponding menu item is selected.

Do NOT save this file as "True BASIC"! That's risky. Save this altered binary file as "True BASIC2" and keep it in the same drawer as the regular True BASIC. This avoids the need to duplicate the startup files, etc., and bails you out if you messed up.

From a shell:

```
[illegible]
[illegible]
[illegible]
```

Now go into the TB:TBDo drawer and make a copy of the FB file, renaming it as FBASIC (the 6 letter name we used in the TB menu).

We are all set. Double click on the new program called "True BASIC2". A new True BASIC editor activates. Look at the pull down menus. In the 'Do' program menu is one called "Do FBASIC". It also has an indicated hot key (right-Amiga-D).

Now load or type and edit an F-BASIC program within this fast, easy, and powerful editor. Use right-Amiga-S to save the text file before compiling, and right-Amiga-D to compile and run your F-BASIC programs. If they croak, you get told the place where. Just type "to linenumber" (to 322, whatever) to go to the bad line. The SDBC needed files are there and can also be used from within the editor with similar strategy.

**What F-BASIC does, it does very very well. It does overscan and AGA graphics. It does many neat things directly that other languages require you to do with nasty add on support.**

Load your s:Shell-Startup file and add this line:

```
[illegible]
```

When you open a shell and type FB: you are in the F-BASIC drawer if FB: was assigned to your F-BASIC directory. Type TB! and you open your new editor in the FB: path as the editor defaults to the path that beckoned it.

The True BASIC editor allows macros to be assigned to keys and saved as key files. 'Do' routines can read and modify your files or search for common errors or replace tokens for expanded commands in exact form, etc.

The very low cost of True BASIC makes this all rather practical. Less than the cost of phone support.

## FC_Do    F-BASIC Compile Only

```
! 'FC_Do'     F-BASIC Compile Only      Written by Roy
M. Pizzo

! Compile and save as 'FC' in the 'TBDo' drawer.
! This is a 'Do' program called from within the True
BASIC editor.
! Save the file to the 'FB:' directory
! At the command window type "do FC, filename" and
press the editor.
! F-BASIC is case sensitive so use correct case.
! The True BASIC editor will launch FB:FB with the
debugger option.
! If the file screws up, the debugger leaves a text file
read by
! this Do program
! The Do program returns to the editor with the line
number that failed.
!
! At the command window type "do <linenumber>"

EXTERNAL

sub BasicCompile(line$(),FileName$)
   LIBRARY "TBDo:library/Amiga.lib*"

   let FileName$ = Trim$(FileName$)

   if FileName$ = "" then
      ! TRY NAME OF CURRENTLY LOADED FILE
      ask name FileName$
      when error in
       open #100: name "FB:" & FileName$, organization
TEXT, CREATE OLD
        close #100
       use
        close #100
       cause error 1, "Use: Do FB, filename"
      end when
   end if


   when error in
        when error in
         call CLI("Delete FB:" & FileName$ & ".ERR")
        use
        end when

       call CLI("FB:FB " & FileName$ & " opt-d")
       sound 1000, 01

        let ErrName$ = "FB:ERR.LIST"
        let count = 0

          WHEN ERROR IN
           open #100: name "FB:" & FileName$ & ".ERR",
organization TEXT, CREATE OLD
             set #100: POINTER BEGIN

           do while more #100
              line input #100: e$
              let count = count + 1
           loop

           let ErrName$ = "Error at " & str$(count) & ".
Assembler error: " & e$
           sound 2000, 10

          USE
          END WHEN
          close #100

         !                    ------ edit running part
         ! call CLI("FB:" & FileName$ & " ...")
         !--------
      use
         cause error 1, ErrName$ & " " & FileName$
         exit sub
      end when

      cause error 1, "Done: " & ErrName$
end sub
```

## FR_Do    F-BASIC Compile & Run

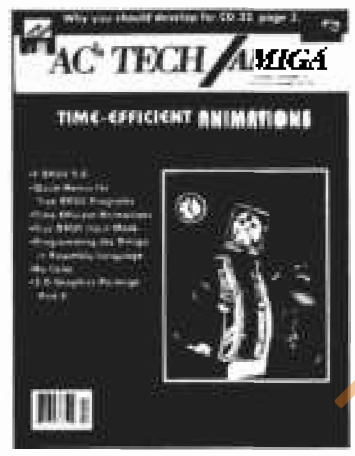
```
! 'FR_Do'     F-BASIC Compile & Run      Written by Roy
M. Pizzo

! Compile and save as 'FR' in the 'TBDo' drawer.
! This is a 'Do' program called from within the True
BASIC editor.
! Save the file to the 'FB:' directory.
! At the command window type "do FR, filename" and
process that error.
! F-BASIC is case sensitive so use correct case.
! The True BASIC editor will launch FB:FB with the
debugger option.
! If the file screws up, the debugger leaves a text file
read by
! this Do program.
! The Do program returns to the editor with the line
number that failed.

! At the command window type "do <linenumber>"
```

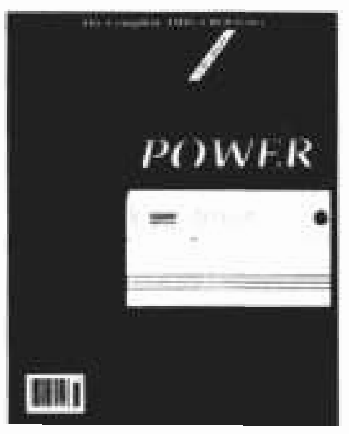
# Three ways to make your life easier:

*Amazing Computing For The Commodore Amiga* is dedicated to Amiga users who want to do more with their Amigas. From Amiga beginners to advanced Amiga hardware hackers, AC consistently offers articles, reviews, hints, and insights into the expanding capabilities of the Amiga. *Amazing Computing* is always in touch with the latest new products and new achievements for the Commodore Amiga. Whether it is an interest in Video production, programming, business, productivity, or just great games, AC presents the finest the Amiga has to offer. For exciting Amiga information in a clear and informative style, there is no better value than *Amazing Computing* .




## AC's TECH AMIGA

*AC's TECH For The Commodore Amiga* is the first disk-based technical magazine for the Amiga, and it remains the best. Each issue explores the Amiga in an in-depth manner unavailable anywhere else. From hardware articles to programming techniques, *AC's TECH* is a fundamental resource for every Amiga user who wants to understand the Amiga and improve its performance. *AC's TECH* offers its readers an expanding reference of Amiga technical knowledge. If you are constantly challenged by the possibilities of the world's most adaptable computer, read the publication that delivers the best in technical insight, *AC's TECH For The Commodore Amiga.*




## AC's GUIDE AMIGA

*AC's GUIDE* is a complete collection of products and services available for your Amiga. No Amiga owner should be without *AC's GUIDE*. More valuable than the telephone book, *AC's GUIDE* has complete listings of products, services, vendor information, user's groups and public domain programs. Don't go another day without *AC's GUIDE*!

**Live better with Amazing Computing**

**1-800-345-3360**

As reviewed by AC Tech #3.4 and Amiga World Aug. 93

# The LANGUAGE For The Amiga!

One Amiga language has stood the test of time.

This new package represents the fourth major upgraded release of F-Basic since 1988. Packed with new features 5.0 is the fastest and fullest yet. The power of C with the friendliness of BASIC. Compatibility with all Amiga platforms through the 4000 ...compiled assembly object code with incredible execution times... features from all modern languages, an AREXX port, PAL and ECS/AGA chip set support...Free technical support... This is the FAST one you've read so much about!

**F-BASIC 5.0™**

Supports DOS 1.3, 2.0, 2.1 and 3.0

**F-BASIC 5.0™ System $99.95**
Includes Compiler, Linker, Integrated Editor Environment, User's Manual, & Sample Programs Disk.

**F-BASIC 5.0™ + SLDB System $159.95**
As above with Complete Source Level DeBugger.

Available Only From: DELPHI NOETIC SYSTEMS, INC **(605) 348-0791**
P.O. Box 7722 Rapid City, SD 57709-7722
Send Check or Money Order or Write For Info. Call With Credit Card or C.O.D.
Fax (605) 343-4372 Overseas Distributor Inquiries Welcome

```
[illegible]

sub BasicCompile[illegible]
  LIBRARY "[illegible]"

  local [illegible]

  if [illegible] then
      [illegible]
    [illegible]
    when error [illegible]
      open #100: [illegible] organization
TEXT, CREATE [illegible]
      close #100
    use
      close #100
      cause error 1, "[illegible]"
    end when
  end if

  when error in
   call [illegible] & ".EXT"
  use
  end when

  [illegible]
  close [illegible]

  local [illegible] = [illegible]
  local [illegible] = 0

  WHEN ERROR IN
   open #100: name "[illegible]" & [illegible] & [illegible],
organization TEXT, [illegible]
      [illegible]

    do while [illegible] #100
       line input #100: s$
       let [illegible] = [illegible]
    loop

    let [illegible] = "[illegible]" & [illegible] & "
[illegible]"
    close #100

  USE
   [illegible]
   close #100

   call "[illegible]" & [illegible]
  use
   cause error 1, [illegible]
   exit sub
  end when

  cause error 1, "[illegible]" & [illegible]
end sub
```

✔

**Complete source code and listings can be found on the AC's TECH disk.**

***Please write to:***
***Roy M. Nuzzo***
***c/o AC's TECH***
***P.O. Box 2140***
***Fall River, MA 02722***

# Examining Compression

BY DAN WEISS

Compression is a hot topic everywhere today. Computer users want to compress the files on their hard drives to save space. Cable companies want to compress their signals so that they can send 500 channels over the cable they are currently sending 50 channels. Graphic programs have new and even more powerful tools of compression with the JPEG and MPEG standards. Soon, they tell us, we will be able to fit 72 minutes of full motion video and sound on a CD where today we can only fit 72 minutes of sound. Everywhere you look compression is in use. But what is it and how does it work? Lets take a look at compression, by looking at two popular methods used.

## A look at common compression techniques

### Put the squeeze on

Compression in the computer sense is taking a file and from it creating a smaller second file that can be used to recreate the first file. There are many ways to do this. All methods come down to one simple idea: replace a large amount of information with a small amount of information by finding things that are common in the data. As an example the file consisting of

[illegible]

could be compressed as:

[illegible]

Where the number preceding the letter tells how many times that letter should be repeated. As you can see this works out pretty good.

except when there is a single letter, then the code takes up more space than the original letter. In fact if you had a block of text that very few letters were next to a similar letter (like this article) then the "compressed" file would be much bigger than the original. In compression "lingo" this is know as the degenerate case, and most algorithms have some such worse case situation. In this algorithm we "let the air out" of the file by counting runs of information that are the same.

Counting the runs of information is known as "Run Length Encoding" or RLE for short. RLE is used in the IFF ILBM standard and is very good where there are long runs of the same value. For this reason a two color picture (black and white for instance) compresses very well where a 24 bit picture (approximately 16.8 million colors) does not. Of course if the whole 24 bit image was on a particular color then it would compress very well, but usually they are not.

Another advantage of RLE is that it executes quickly and is easy to implement. The following is a block of pseudo code that implements an RLE compressor.

```
/* Rule compressor */
curRun = NULL;
curRunCount = 0;
while (not end-of-infile)
{
		curChar = fscanf(infile,"%c");
		if (curChar != curRun)
		{
				if (curRunCount != 0)
				{
						fprintf(outfile,"%c%c",curRunCount,curRun);
				}
				curRunCount = 1;
				curRun = curChar;
		}
		else
		{
				curRunCount ++;
		}
}
```

The only problem with this code is that it assumes that the program that decompresses the file can tell a number from a piece of data. In the previous example let's replace the letter 'A' with 1, 'B' with 2 and 'C' with 3. The data then becomes:

"[illegible]"

and the compressed version becomes:

[illegible]

This is clearly a problem since you can't tell the numbers from the data. We can get around this with the following convention. The first byte of the file will always be a count byte. The second byte will always be a data byte. This means that we can only do runs of at most 255 characters before we must define a second run of the same character. This way, the odd bytes in the file are count bytes and the even are data bytes.

Modifying the above code we get:

```
/* RLE compressor */
curRun = NULL;
curRunCount = 0;
while (not end-of-infile)

		curChar = fscanf(infile,"%c");
		if (curChar != curRun)
		{
				if (curRunCount != 0)
				{
						fprintf(outfile,"%c%c",curRunCount,curRun);
				}
				curRunCount = 1;
				curRun = curChar;
		}
		else
		{
				curRunCount ++;
				if (curRunCount > 255)
				{
						fprintf(outfile,"%c%c",curRunCount,curRun);
						curRunCount = 0;
				}
		}
}
```

Now we can write the decompressor. The pseudo code would be:

```
/* RLE decompressor */
while (not end of infile)
{
		fscanf(infile,"%c%c",runCount,runChar);
		for (i = 0; i < runCount; i++)
		{
				fprintf(outfile,"%c",runChar);

}
```

As you can see the decompressor is trivial, which is a desirable attribute. In most cases it is more important to be able to decompress quickly than to be able to compress quickly. In the case of high speed animation (30 frames per second) compression is used as a way to get the image to/from the hard drive (or CD-ROM) quickly (by having less data to load). Once the data is loaded it must be expanded quickly. This algorithm does that.

But it needs a better ability to handle data that is nearly random. As mentioned in the beginning the case of every character being different than the last one would result in a doubling of the file. What we need is a way to block out these runs of randomness. Taking a page from the IFF ILBM implementation of RLE we will do the following:

If the high bit is set on a count byte then clear the bit and copy the number of bytes indicated directly from the file.

If the high bit is not set on a count byte then copy the next byte the number of times indicated.

This reduces the maximum run to 127 but handles random cases much better. The modified compressor pseudo code looks like this:

```
/* Rule compressor with random support */
curRun = NULL;
curRunCount = 0;
randomString = NULL;
randomCount = 0;
while (not end-of-infile)
{
		curChar = fscanf(infile,"%c");
		if (curChar != curRun)
		{
				if (curRunCount != 0) /* ending a run */
```



DJComputers.cz

```
		{
			fprintf(outfile,"%c%c",curRunCount,curRun);
			curRunCount = 1;
			curRun = curChar;
		}
		else /* just beginning or in a random block */
		{
			if (randomCount != 0)
			{
				if (randomString[randomCount-1] ==
curChar)
				{
					/* second char in run
*/
					curRunCount = 2;
					curRun = curChar;

randomString[randomCount-1] = 0;
					randomCount = 128; /
* set high bit */

fprintf(outfile,"%c%s",randomCount,randomString);
					randomCount = 0; /*
no random data now */
				}
				else
				{
					/* in another
random character */
					if (randomCount ==
127)
					{

randomString[randomCount] = 0;

randomCount |= 128; /* set high bit */

fprintf(outfile,"%c%s",randomCount,randomString);

randomCount = 0;
					}

randomString[randomCount] = curChar;
					randomCount ++;
					curRun = curChar;
				}
			}
			else /* just starting out, treat as random
*/
			{
				randomString[randomCount] =
curChar;
				randomCount ++;
				curRun = curChar;
			}
		}
	}
	else
	{
		curRunCount ++;
		if (curRunCount > 127)
		{
			fprintf(outfile,"%c%c",curRunCount,curRun);
			curRunCount = 1;
			curRun = curChar;
		}
	}
}
```

The code has grown a bit, but we now handle random runs of up to 127 characters at a time. In the example string this change the string from:

```
[illegible]
```

to:

```
[illegible]
```

Which doesn't look smaller because we are writing out the numbers, but actually uses 5 bytes less. If the data were more random the savings would be greater. Looking at the degenerate case of total randomness we only add one byte for every 127 characters. If the file has only one run of four characters out of every 127 then we end up with no net enlargement. Any more or larger runs result in some compression. The best case is when each run is 127 characters long resulting in a 63.5 to 1 compression ratio! This of course does not happen in the real world, but is fun to think about.

The de-compressor does not get much more complicated with the new rules. The pseudo code becomes:

```
/* RLE decompressor with random support */
while (not end-of-infile)
{
	fscanf(infile,"%c",runCount);
	if (runCount > 128) /* random */
	{
		runCount -= 128; /* clear the top bit */
		fread(randomString,runCount,1,infile);
		randomString[runCount+1] = 0;
		fprintf(outfile,"%s",randomString);
	}
	else /* a run */
	{
		fscanf(infile,"%c",runChar);
		for (i = 0; i < runCount; i++)
		{
			fprintf(outfile,"%c",runChar);
		}
	}
}
```

## Look it up in a dictionary

One of the advantages of RLE is that it can adapt to any data. This is because it is only worried about the data one byte at a time. This is also a disadvantage when the data tends to repeat itself over a range of more than one byte. For instance the string:

```
[illegible]
```

would not compress at all under the RLE algorithm because every character is surrounded by different characters. On the other hand you can see that the string is simply the words "IN" and "OUT" repeated over and over. What would be great is if we could create a special compression scheme where "IN" was replaced with 1 and "OUT" with 2. Then the string would be:

```
[illegible]
```

A savings of 18 vs 48, better than 2.5 to 1 compression. If the string were longer, then the savings would be bigger. We could also apply the RLE algorithm to the compressed file and gain a little more. The approach of using short codes to replace chunks of the file is known as dictionary based compression (from the fact that the codes are looked up). Dictionary based compression can be very powerful since it can look at larger parts of the file. The problems with this method are two fold. First you must send a copy of the dictionary with the file which increases the size of the file losing some of the benefit of the compression, and secondly choosing and building a dictionary is not trivial and takes time.

The first problem can be alleviated if you come up with a static dictionary that can be used for many files. When using a static

dictionary it only needs to be built into the compressor and decompressor, not sent. This is the idea used by fax machines. When defining the Group 3 fax standard thousands of faxes were analyzed to find typical runs of white and black dots. When they were finished, a static dictionary was released, and built into every Group 3 fax machine. Because there is a very high chance that noise on the phone line could scramble the data, each line of the fax is treated as a separate "file". In a perfect world the whole fax would be one stream of data, but if a code gets changed then count bytes could be read as data and vice versa resulting in chaos. Treating each line separately limits the damage to a single line.

It would seem that every fax is very different and that a simple RLE algorithm would be better but the dictionary based compression works very well, but only for faxes. If you try to apply the fax dictionary to regular data, the results are not as good. As you would expect this is typical of all dictionary based compression methods. An optimal dictionary for one file is not optimal for another.

How then do you build an optimal dictionary? In the case where there is a logical unit of information, like words in a sentence, a dictionary can be built from these units, much like in the "IN/OUT" example. In situations where the data appears random what you choose for the entries in the dictionary don't matter. Don't matter? you say. No, it doesn't really matter. Pioneering work by Abraham Lempel and Jacob Ziv (the L and Z of LZW compression) showed in the late seventies that you can create a useful dictionary on the fly by looking at the file.

## Lempel, Ziv and Welch

LZW (Lempel - Ziv - Welch) compression is the backbone of much of the general purpose file compression today. The modem V42.bis compression method is based on LZW and the GIF and TIFF graphic formats use it as well. The idea is deceptively simple yet represents a major breakthrough. The basic algorithm works as follows:

Create a dictionary array with some number of entries (2K to 4K entries is typical). Assume that the table starts with entry 256 because entries 0 to 255 are assumed to contain themselves (ie. entry 230 contains 230). Read the first two bytes in the file. Place the combination of the two bytes into the dictionary as the first entry. Since there was not a match for the pair in the dictionary originally, output the first byte and shift the second one down. Read the next byte. Does the buffer now hold a combination that is in the dictionary. If not, do as the first time. If it does match a combination in the dictionary, then keep reading data until you can no longer make a match. Then make a new code out of all the data and output the last code that was matched. Clear the buffer down to the last character read and start again.

What this does is continue to build longer and longer codes based on what has come before. This is perfect for picking up runs of characters that repeat, like words. It may seem very inefficient to add a new code into the dictionary for every character or group of characters encoded but it has two advantages. The first is that you never know when a sequence of data will reoccur. By building a large table you stand a very good chance of catching the same sequence again. Using this method you can even catch longer sequences like repetitive phrases (such as "I have a dream" from Martin Luther King Jr.'s famous speech). It also is very good at encoding long runs of the same data. The repeating string of XO (as in XOXOXOXOXOXOXOXO) would not be compressed by RLE encoding. But the LZW version would first save the X then the O, then the codes for XO, XOX, OX, OXO and finally XO. All together the data would be reduced to seven codes, a significant savings.

The second reason to encode all of the combinations is so that you do not have to send the dictionary along. Remember one of the disadvantages of dictionary based compression is having to send the dictionary. By using the a well defined method for creating the dictionary from the data you can create the dictionary from the compressed data and codes as well. Let's look at how.

**Soon, they tell us, we will be able to fit 72 minutes of full motion video and sound on a CD where today we can only fit 72 minutes of sound.**

DJComputers.cz

### How's it done?

Going back to our first example we start with:

[illegible]

The first two characters are read in. Since no codes exist yet, the first 'A' is output and the combination 'AA' is made entry number 256. Now the third 'A' is read in. Combined with the second 'A' it makes 'AA' which is code 256, so we read in another byte hoping to find an even longer match. We don't so we output the code 256 in place of 'AA', take the current buffer of 'AAA' and make it code 257. Then we remove the first two 'A's since they have been output and read in the next character. Reading in the fifth character we get 'AA' in the buffer. Since we have a match for 'AA' we read the next character hoping for a longer match. The sixth character is an 'A' so the buffer is now 'AAA'. This matches code 257 so we add another character. The seventh character is also an 'A'. There is no code for 'AAAA' so we output 257, create code 258 and remove the code 'AAA' from the buffer leaving 'A'. The next piece of data is 'B'. Combined with the 'A' we get 'AB'. There is no code for this so we have to output the 'A' by itself and create code 259. The next 'B' combines to make 'BB'. There is no code for this either so the first 'B' is output and 'BB' becomes code 260. Next is an 'A' the 'B' in the buffer

and the 'A' make 'BA' which has no code. Code 261 is created and the 'B' is output. Next is an 'A'. 'AA' is a code we know so we try for more. 'AAA' is known as well so we continue. 'AAAA' is also known so we try for five 'A''s. We don't have a code for five 'A''s so we output code 258 and create code 262 with five 'A''s.
Let's stop here a minute and make a table of what we have done.

| Input | Output | Code Table |
|---|---|---|
| 'AA' | 'A' | 256 = 'AA' |
| 'AA' | 256 | 257 = 'AAA' |
| 'AAA' | 257 | 258 = 'AAAA' |
| 'B' | 'A' | 259 = 'AB' |
| 'B' | 'B' | 260 = 'BB' |
| 'A' | 'B' | 261 = 'BA' |
| 'AAAA' | 258 | 262 = 'AAAAA' |

As you can see it paid off later for creating the code 258 earlier. At this point any run of 'A''s from two to five characters can be encoded by a single code.

But how does it look from the other end? How does the decompressor recreate the file? Working only from the data in the file we will recreate the dictionary and the file. The compressed file is:

[illegible]

The decompressor starts with an empty dictionary. Reading the first 'A' the decompressor places it in an empty buffer. The next code is [256]. This means that the compression program found a match on the second character. Since the first character is an 'A' and the second and third characters were the same as the first and second, the first two characters must be the same. The decompressor adds a second 'A' to the buffer and creates code 256, which it then outputs and leaves 'AA' in the buffer. The next piece of data is a code as well so again the new character must be an 'A'. Code 257 is created and output. The next piece of data is an 'A'. Since this is not a code, then we know we need to flush the buffer. The code 258 is created and the buffer is flushed except for the 'A' just read in. Finally the 'A' is output. The next character is a 'B'. Again since this is a character code, the code 259 is created and the buffer is flushed except for the 'B', which is output. The next 'B' repeats the process. The code 258 causes the buffer to be loaded with the first character from the code. Code 261 is created from the 'B' and the first 'A' in code 258. Now the entire entry for code 258 is loaded and output.

As you can see, the dictionary is recreated on the fly from the order of the data and the rules used to create it. While the explanations can get a bit lengthy it actually works out to be a straight forward algorithm. It is very good at compressing files that have repetitive data in them. Text files are loaded with repetitive data as are many graphics files. For example, a black and white file that has a gray pattern (every other bit is black) does not compress using an RLE algorithm. But as shown by the "XOXOXO" example earlier, LZW will get better and better at compressing the file until it can take very large chunks at a time. This ability to adapt to the data is what makes it so powerful.

## What's the difference

When we first looked at dictionary compression we noted that the best results come when you understand the data you are compressing. LZW does well even when it knows nothing about the data. But if we look at the data, very often we can "help" the LZW algorithm. The best case of this is called horizontal differencing. If you look at 24 bit pictures, they tend to feature gentle transitions from one color to another unlike Deluxe Paint pictures which offer sharp contrasts. If you think about it the transition from light to dark red is the same as from light to dark blue except for the color. To take advantage of this, we take the data and preprocess it.
Take the value of the first pixel and subtract it from the second pixel. Store the original first pixel and the difference between the first and second as the second pixel. Compute the third pixel by subtracting the original second pixel from the third pixel. Continuing throughout the file you will notice some things. In places where there is a gentle even change, you will end up with long strings of the same value, which compress very well. Gradations of all colors now look the same (and compress with the same codes) because the change data instead of the color data is being recorded. In the case where the colors are random, then the data is random either way. In effect, this becomes a win/win situation for very little effort. The file is easy to reconstitute after it has been uncompressed. Simply take the first pixel and add it to the second. Add the second to third and so on.

## More later

Well that concludes an overview of two of the more popular methods of data compression. There are many others, and there is even more to LZW than what we covered. In another issue of AC's Tech we will look at implementing these and other compression algorithms on 'CX. Until then, keep in touch through this magazine or via internet at danw@ssi.com.

**Complete source code and listings can be found on the AC's TECH disk.**

***Please write to:***
***Dan Weiss***
***c/o AC's TECH***
***P.O. Box 2140***
***Fall River, MA 02722***

# A Date with TrueBASIC

*by T. Darrel Westbrook*

We use calendar dates in many applications. Working with dates requires planning regarding their use internally in the program and how we display or use them in a program. We must check dates entered by application users for accuracy. The checking process must make allowances for the day of the month, whether it's a leap year, and how you would like to display the date. Personally, I dislike cryptic date displays that are difficult to read at a glance. Consider 01-07-93. Is it 7 January 1993 or 1 July 1993? A displayed date (on screen or printed) should be easily read, like 1 Jul 93, and the format should be user selectable. With the ability to determine the day of the week, you can write applications that build calendars, display full date information, preform scheduling functions, etc. This article explains a True BASIC module that supplies these attributes for your programming use.

I designed the True BASIC Date Module, outlined in this article, for maximum flexibility. It returns suitable date information to the calling program for use on screen or on a printed medium. The date information is also usable for sort keys. Additionally, this article highlights a work around for a bug in the Amiga True BASIC language date functions, which is present in both version 1.0 and 2.0 of the language. Finally, I'll discuss error routines and how you can cheaply, code wise, trap a lot of general input errors.

You can use Listing 1 to test the Date Module capabilities. The Global Module subroutines, 'error_reset' and 'make_error', are necessary for the Date Module to function properly and must be including in your programs. Reference lines 93 through 114 for these two subroutines. The Date Module is Listing 2. Line numbers are for reference only.

Throughout this article I will reference a 'date template'. The date template is any combination of day, month, and year expressed as D or DD for day, MM or MMM or MMMM for month, and YY or YYYY for

'02 Jan 94' is the system date, which is a 'Sunday'.
I used a date format of DDMMMYY.

Left: Screen showing the current system date.



DJcomputers.cz

year. The date module converts any case combination of the date template into upper case. Later, I'll discuss what each part of the date template represents. But first, calendar background will provide historical insight behind the structure of the Date Module.

A calendar, according to the Encyclopaedia Britannica, "...is a means of grouping days in ways convenient for regulating civil life and religious observances and for historical and scientific purposes." An ideal calendar would be tied to the movement of the moon phases, seasonal events, and religious holidays. Astrological events, which happen every year but are slightly different each year, are the basis for most religious holidays. For example, the vernal equinox is the basis of the Christian Easter holiday. The vernal equinox occurs when the sun passes northward over the equator. It marks the first day of spring and is generally around 21 March. The exact occurrence of the many Christian holidays use the Easter holiday as a baseline. It is easy to understand why this was important to religious leaders. Other religious holidays, like the Jewish Passover, are determined by their relationship with specific astrological events, like the vernal equinox.

In February 1582, Pope Gregory XIII issued a proclamation that brought the vernal equinox back to 21 March. The proclamation abolished ten days that had accumulated over the past centuries. The Pope added the ten days to 5 October to make it 15 October for the Feast of St. Francis, which occurred on 5 October. The 365.2422 days per year became the new year length. The year length of 365.2422 days per year replaced the old Julian calendar year length of 365.25 days per year (the correct value is 365.24199). This was a difference of 0.0078 days from the old Julian calendar (used since 45 B.C.) and resulted in 3.12 day error for every 400 years. This correction eliminated three out of every four centennial leap years in the Julian calendar. This change is why every centennial year that is evenly divisible by 400 is a leap year. For example, the year 2000 is a leap year, but the centennial year 1900 is not.

Leap years happen every four years (years that are evenly divided by four) besides the centennial leap years. The year 1992 and 1996 are both leap years. If you calculate dates before 15 October 1582, you would need to allow for the changes made by Pope Gregory XIII's proclamation. It could get quite involved. The Date Module uses 1 January 1900 as the cutoff to simplify the program. I don't have any need for dates before 1900, but I didn't want to build too much limitation into the module. The module determines if a year is a leap year in the subroutine (see lines 597 to 611) and sets the leap_flag variable to either one (leap year) or zero (no leap year).

You can use the Date Module to establish a specific date template for use within your programs. You can allow the user to change the date template by using the Change_Date_Format subroutine. For a given date, the module subroutines return the day of the week (Sunday through Saturday) and a day of the week number (1 through 7). The day of the week number is suitable for determining workdays for scheduling programs, calendar generating programs, etc.

The Date Module returns similar information for the month (i.e., numbers 1 through 12 and corresponding names of January through December). Month abbreviations (i.e., Jan, Feb, etc.) are also available for your programming use. The subroutine provides two sortable dates for internal program use. They are the string form of and YYYYDDD. (The date template is the format used by parse_date to arrange the order of day, month and year in the_date. The order of the individual items in the date template is as important as the location of the Ds, Ms, and Ys. The module will accept single or multiple Ds for the day. The subroutine pads a single digit day with a leading blank for a single D character in the date template. For multiple Ds it pads the single digit day with a leading zero. The month format is similar to the day determination.

The date template determines the returned month format and where the parse routine searches the input string for the month. If the template has one or two Ms, it will return the date as a two digit numeric month. If your date template has three Ms, the date returned will have a three character month (like Jun). When the Ms in the template number four or more, then you get the full spelling of the month. A similar operation occurs with the year, but it will only accept two and four digit

Change Template, See Template, System Date, or Date

Above right: The cil overlay.

Right: The Change Template screen.

years. Therefore, the only valid year input is YY and YYYY template. The rtn_date variable is the returned variable in the date template format. If there are other combinations or formats of dates you would like returned to your calling programs, it is easy to modify the Date Module to get exactly what you want. Since the date template determines the contents of rtn_date, you probably shouldn't change it. You can change any of the other returned variables without affecting the Date Module performance or you could add another variable to the argument list of the parse_date subroutine.

The parse_date subroutine (lines 394 through 510) will recognize program inputs that contain date template delimiters. A date delimiter is a printable character that separates the day, month, and year. The module uses the delimiter to format the returned date values (i.e., rtn_date $). The most common delimiter is the dash (-) or minus sign between the day, month, and year. For example: the program code converts 1 Jul 93 to 01-07-93 for a date template of DD-MM-YY. The Date Module will recognize any printable character except an alphabet letter or a number as a date template delimiter. If you have several delimiters in a given template, the program will recognize them all, but will use only the last one in the template string as the delimiter for the module. When you use the Date Module in a True BASIC program, lines 202 through 226 initialize the module. The SHAREd variables (lines 202 through 215) are self-explanatory. The Max_Day subroutine and the date function are PRIVATE to the module (lines 216 and 217). These two are not callable outside the Date Module. I did this to prevent changing the year, max_day, m_factor, and leap_flag outside of module control. If you want to address these procedures outside the limits of the module, remove the PRIVATE statements.

Lines 218 through 225 initialize Date Module variables. The Change_Date_Format? subroutine (line 284) initializes the date template variables length, lengthm, lengthy, delimiter, user_date_format, and order_date_string. I used these variables throughout the module as module specific global variables. It cuts down a lot of code overhead when you SHARE variables within a module. I have included a small subroutine, Get_Date_Format, to return the current template to the calling program.

If you want to initialize your Date Module with a different template, change line 224 to the template you want to use. You must also change the order_date_string variable, line 225. I used the order_date_string variable to resolve conflicts that occur from user input. It is a four character string, with a blank for the first character. The following three characters must be one each of D, M, and Y. The DMY characters help module subroutines make decisions regarding user input when it doesn't exactly match the date template or other input parameters. The parse_date subroutine, line 394, uses the order_date_string to resolve a user input date to the date template. This subroutine parses any date input (like user input) that does not originate from the computer system. The date module gets the computer system date from the System_Date subroutine. You must be cautious when you use the Amiga True BASIC date functions. These date functions have a software bug that can make your system date seem inaccurate.

True BASIC has two built in date functions. They are date and date$. The date function returns the current machine date in the form of YYDDD, which is a Julian date format, and date$ uses the YYYYMMDD format. The Amiga True BASIC version does not recognize leap years.

It manages dates well until 29 Feb of a leap year. On that day the language will return 1 Mar of the leap year. From then on the date returned by True BASIC is one day off throughout the remainder of the leap year. Both functions (date and date$) are off by one day.

On 31 Dec of the leap year, True BASIC gets confused and finally realizes that something is wrong. It returns the date YYYY1200 for the date$ function and YY000 for date function. This is the only instance when True BASIC returns a zero for the day. When the user enters a non leap year date or the system date does not reflect a leap year, the language will return valid dates. The code from the date$ function (lines 259 to 293) in Listing 2 corrects the True BASIC software bug. The bug is not present in the IBM version of the language. Delete lines 215 to 274, if you use this code for the IBM machines. Although the IBM True BASIC version does not need this code, the code will function as outlined in this article on an IBM compatible machine.

You can test this software bug by using a utility that allows you to change the Amiga system date like TimeSet 2.0, by David Holt. If you don't have one of these Public Domain date/time setting programs, then use the CLI program date. From a CLI, type:

```
date 29 Feb 92
```

which is a leap year. Just to see if the system accepted the date, type in date which should display Saturday 29-Feb-92 plus the cur-

Left: The Date Module screen.

DJComputers.cz

run time. Use the Amiga's multi-tasking capabilities and have True BASIC up and running. From True BASIC command input screen (press F2 while in the True BASIC editor), type:

print date$

which will print out 19920301, which translates to 1 Mar 92. You can check the YYYYMMDD True BASIC output by setting the system date to 31-Dec 92. Beware that the AmigaDOS date program will not accept dates before 1 Jan 78. This is the date the software recognizes as the base line of its existence. The Date Module does not share this limitation. It will accept dates between 1 January 1700 to 31 December 2199.

The Date Module Date_Data subroutine returns the following information to the calling program: year, the numeric year in YYYY format; month, the numeric month (i.e., 1 through 12); day, numeric day of the month; julian, julian date in the YYYYDDD format; month$, full alphanumeric month name, like June; m_abrev$, abbreviation for the month, like Jun; dow_factor, numeric day, 1 to 7 (Sunday to Saturday); dow_name$, alphanumeric day of the week, like Sunday; mn_date$, date based on the module date template; sort_date$, YYYYMMDD format.

As you can see, any program which uses dates can use the Date Module. The program also uses a novel means to pass error information back to the calling program without creating a runtime error.

True BASIC has three global variables that it uses for error reporting. They are EXLINE$, EXTYPE$, and EXTYPE. These are normally null and zero, but when an error occurs they are set to the line number, the type of error and the error number. Appendix C of the True BASIC Reference Manual lists all the language built-in error information. Since the language is expandable, there are commands that allow you, the programmer, to create your own error traps and handlers.

To create your own error traps you use the WHEN ERROR IN ... USE ... END WHEN structure. If you cause an error, by using the CAUSE ERROR or CAUSE EXCEPTION commands within this structure the program will not experience a fatal runtime error and stop the program. By using the subroutines error_reset and make_error, you can cause an error of your choosing, then check the EXLINE$, EXTYPE$, and EXTYPE variables in your calling program. If EXTYPE is anything but zero, an error of some type has occurred. Using these global variables releases you from passing error messages back and forth from calling routines to the module and back again.

Line 240 is an example of how this module handles error management. If the value of the system date is zero, then the system date is not

→

Selected as the best professional productivity software at the last two North American Amiga Developers' Conferences, the SAS/C Development System now includes C++.
If you are currently using another commercial C compiler, call now for details on our special trade-in offer!
For more information and to order, call SAS Institute at 919-677-8000, ext. 7001.
SAS and SAS/C are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. Other brand and product names are registered trademarks or trademarks of their respective holders.
Readers' Choice Award
Amazing Computing
1993
SAS®
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513

set. There is no error built into the language that reports that the system date is not set. Line 24 creates an error by passing an error number and an error message to the make_errors subroutine. This subroutine sets EXTYPE to 124 and the EXTYPE$ to "Computer system date is not set." The EXLINE$ variable is set to either line 104 or line 109, depending on the magic variable. Program flow then returns to the calling routine which only needs to check the EXTYPE to determine if the system date is set. Line 325 and lines 506 to 509 are other examples of using this technique to handle errors in your program.

The True BASIC Date Module in this article should be a valuable addition to your programming library. The Date Module will decrease your programming time and provide flexibility in handling date variables. It and the other routines are valuable tools to add to your True BASIC library.

## Listing One

```
1  [illegible] program for Date Module
   [illegible] 1993
2  [illegible]
3  [illegible]
4  [illegible]
5  [illegible]
6  [illegible]
9  CLEAR
10 CALL Center("Date Format [illegible]",4,3)
11 CALL Center("Change [illegible]
   [illegible]
   [illegible] KeyInput (10,2$)
   [illegible]
14 [illegible]
15 [illegible]
16       DO
17          CLEAR
18          CALL Center("Enter new date [illegible]",4,3)
19          [illegible]
20          [illegible]
21          [illegible] Change Date Format [illegible]
22          IF [illegible]
23          CALL Center("[illegible]",8,2)
24          CALL Center("Error!" [illegible])
25          [illegible]
26       LOOP
27       [illegible]
28       IF [illegible]
29          CALL [illegible]
30          CALL Center("Date [illegible]",4,3)
31          [illegible]
32          CALL Center("Press any key to continue" [illegible])
33          [illegible]
34          [illegible]
35          CLEAR
36       END
37    CASE "[illegible]"
38          CLEAR
39          [illegible]
40          CALL Center("[illegible]
   [illegible])
41          CALL Center("Press any key to continue" [illegible])
42          [illegible]
43          [illegible]
44          CLEAR
45    CASE "SYSTEM DATE"
46          CLEAR
47          CALL System_Date [illegible]
48          [illegible]
49          CLEAR
50          CALL Center("[illegible]
   [illegible]
51          CALL Center("[illegible] Date Format [illegible]
   [illegible])
52          CALL Center("[illegible]
   [illegible])
53          CALL Center("Press any key to continue",10,1)
54          CALL [illegible]
55          [illegible]
56          CLEAR
57    CASE "[illegible]"
58          CALL [illegible] Format [illegible]
59          DO
60             CLEAR
61             CALL Center("[illegible]
   [illegible]
62             CALL KeyInput (10,[illegible]$)
63             IF [illegible]
64             CALL parse_date [illegible] month,day)
65             CLEAR
66             IF [illegible]
67                [illegible]
   [illegible]
68                CALL Center("You [illegible]
   [illegible]
69                CALL [illegible] the module using
   the [illegible]
70                CALL [illegible]
   [illegible]
71                CALL Center("Year, [illegible]
   [illegible]
   [illegible]
72                CALL [illegible] day [illegible]
   [illegible]
73                CALL Center("[illegible]
   [illegible]
74                CALL Center("Day of the week [illegible]
   [illegible]
75                CALL Center("[illegible]
   [illegible]
76                CALL Center("[illegible]
   [illegible]
77             ELSE
78                CALL Center("[illegible]",4,3)
79                CALL Center("[illegible]",8,[illegible])
80             END IF
81             CALL Center("Press any key [illegible]",[illegible])
82             CALL [illegible]
83             [illegible]
```

```
84              CLEAR
85           [illegible]
86      CASE else
87           [illegible]
88           EXIT DO
89      END SELECT
90 LOOP
91 END
92 EXTERNAL

93 MODULE Global
94 OPTION BASE 1

95 SUB [illegible]_reset
96      WHEN error in
97           [illegible] EXCEPTION [illegible] ! reset EXTYPE error [illegible]
98      USE
99      END WHEN
100 END SUB    ! end of [illegible]_reset

[illegible] SUB make_error(n,msg$)
102      IF len(msg$) = 0 then
103        WHEN error in
104             CAUSE ERROR n
105        USE
106        END WHEN
107      ELSE
108        WHEN error in
109             CAUSE ERROR n,msg$
110        USE
111        END WHEN
112      END IF
113 END SUB        ! end of 'make_error'
114 END MODULE     ! end of 'Global'

115 !
116 ! support subroutines
117 !
118 SUB KeyInput (row,r$)
119      SET CURSOR "OFF"
120      CALL buffer
121      LET col = 40   ! start at [illegible] of the screen
122      SET CURSOR row,34
123      PRINT repeat$([illegible]) ! [illegible] the input box
124      LET r$ = ""              ! initialize return string
125      DO                       ! forever loop
126        SET COLOR 3            ! [illegible] color
127        SET CURSOR row,int([illegible] len([illegible]) - [illegible])
128          [illegible]
129          DO
130             GET KEY keycode
131             SELECT CASE keycode
132             CASE 8,13,32 to [illegible]
133                  IF keycode >= 32 and keycode <= 126 then
134                    LET t$ = chr$(keycode) ! printable char
135                  ELSE
136                    LET t$ = ""          [illegible] character
137                  END IF
138                  EXIT DO
139             CASE else
140                  ! try again
141             END SELECT
142          LOOP
143          SELECT CASE keycode
144          CASE [illegible]              ! 32 and [illegible]
145               IF col = 40 then
146                  SOUND 100,.5
147                  ! sound [illegible]    [illegible]
148               ELSE
149                  LET r$ = r$[1:len(r$)-1] ! test the last
input to string
150                  SET COLOR 1 ! change color [illegible]
151                  SET CURSOR row,col
152                  PRINT repeat$([illegible])
153                  SET CURSOR row,int([illegible] len(trim$(r$))/2)
! [illegible] the current string
154                  PRINT r$ & [illegible]
155                  LET col = col + 1  ! increment [illegible]
156               END IF
157          CASE [illegible]                     ! CR
158               IF [illegible]
159                  IF [illegible] = -1 and [illegible] = 0 then LET
r$ = ""
160               ELSE
161                  ! [illegible]
162               END IF
163               EXIT DO     ! exit if character is [illegible]
164          CASE 32 to 126           ! printable characters
165               SET COLOR 1
166               SET CURSOR row,col
167               LET r$ = r$ & t$ ! add [illegible]
168               SET CURSOR row,int([illegible] len(trim$(r$))/2)
169               PRINT r$
170               IF len(r$) = 20 then EXIT DO ! [illegible]
max string length
171               LET col = col + 1 ! increment col counter
172          CASE else
173               ! [illegible] selected which is not allowed by
[illegible]
174          END SELECT
175      LOOP      ! end of forever loop
176      SET COLOR [illegible]
177      SET CURSOR row,int([illegible] len(r$)/2)
178      [illegible]
179      LET r$ = trim$(r$)
180 END SUB ! end of 'KeyInput'
181

182 SUB Center(text$,row,Pen Color)
183      SET COLOR Pen Color      ! change text color
184      SET CURSOR row,int((80 - len(trim$(text$)))/2)
185      PRINT text$
186 END SUB                       [illegible] 'Center'

187 SUB Buffer
188      DO                       ! clear keyboard buffer
189        IF key input then
190           GET KEY k           ! get any keyboard input
191           GET MOUSE j,k,l     ! get mouse input [illegible]
192        ELSE
193           GET MOUSE j,k,l     ! get mouse [illegible]
194        END IF
195        [illegible] and [illegible] key [illegible] then EXIT SUB
196      LOOP
197 END SUB                       ! end of Buffer
```

zconnects.cz

## Listing Two

```
198 MODULE Date
 99 ! Date Module for True BASIC, Amiga Version [illegible]
200 ! [illegible] by [illegible] Westbrook, 1993
201 !
202 SHARE month$ ! [illegible] January, February, [illegible]
203 SHARE days ! day of the week
204 SHARE max_days ! variable
205 SHARE max_day1$ ! non leap year max days in a month
206 SHARE max_day2$ ! leap year max days in a month
207 SHARE n_factor$ ! [illegible] used in calculation of the
day of the week
208 SHARE leap_flag ! [illegible] = to 1, leap year, [illegible] for non leap
year
209 SHARE [illegible] ! [illegible] of [illegible] in the user_date_format$
210 SHARE [illegible] ! number of M's in the user_date_format$
211 SHARE lengthd ! number of D's in the user_date_format$
212 SHARE user_date_format$ ! date format, [illegible], [illegible],
[illegible], etc.
213 SHARE order_date_string$ ! [illegible] order of D M Y, for
the returned date
214 SHARE delimit_flag ! [illegible] is used in a date
215 SHARE delimiter$ ! single character to separate date [illegible]

[illegible] max_day
[illegible] date$

218 ! initialize variables
219 !
220 LET max_day1$ = "312831303130313130313031"
221 LET max_day2$ = "312931303130313130313031"
222 LET month$ = " January February March April
May June July August September October November
December"
223 LET days$ = " Sunday Monday Tuesday Wednesday
Thursday Friday Saturday"
224 LET user_date_format$ = "[illegible]"
225 LET order_date_string$ = "[illegible]"

226 CALL Change_Date_Format(user_date_format$, [illegible]
variables)

227 ! [illegible] and functions

228 SUB Get_Date_Format(string$)
229 LET string$ = user_date_format$
230 END SUB

231 SUB System_Date(num_date$,day_name$,julian$)
232 LET x$ = date$
233 LET year = val(x$[1:4])
234 LET month = val(x$[5:6])
235 LET day = val(x$[7:8])
236 CALL [illegible]
[illegible]
237 END SUB

238 DEF date$
239 ! [illegible] 123, computer system date is not set.
240 IF [illegible] = 0 then
241 CAUSE [illegible] "Computer system date is not
set."
242 EXIT DEF
243 END IF
244 CALL error_trap ! reset TB global error [illegible]
245 !
246 ! error trap for True BASIC date$ function bug
247 !
248 IF date$[illegible] then ! [illegible] leap year, TB DATE$
function
249 LET day = 31 ! 31 Dec of leap year is returned as
[illegible]
250 LET month = 12 ! [illegible] being the entry year + 1
251 LET year = val(date$[1:4])-1
252 CALL Max_Day(year)
253 ELSE ! date is not 31 Dec of a leap year
254 LET day = val(date$[7:8])
255 LET month = val(date$[5:6])
256 LET year = val(date$[1:4])
257 CALL Max_Day(year)
258 IF leap_flag = 1 then ! this is a leap year
259 SELECT CASE month
260 CASE [illegible]
261 ! error does not [illegible] 29 February,
262 ! True BASIC believes [illegible]
263 ! leap year is 1 Mar of that year
264 CASE 2
265 IF day > 28 then [illegible]
266 CASE [illegible]
267 [illegible] dates
268 END SELECT ! end of CASE month
269 END IF ! end of "IF leap_flag = 1" ...
270 END IF ! end of "IF date$[1:8] ="[illegible]" ...
271
272 ! end of [illegible] version 2.0
273 ! error trap for known date$ function bug.
274 !
275 LET [illegible] = using$("####",year) & using$("##",month)
& using$("##",day)
276 [illegible]
277 LET day = day -
278 IF day = 0 then
279 LET month = month - 1 ! change month first, so
the day will be correct
280 LET day = val(max_day$[month*2-1:month*2])
281 END IF
282 END [illegible]
283 END DEF ! end definition [illegible]

284 SUB Change_Date_Format(string$)
285 ! [illegible] 124, "Date format string [illegible] string$
& [illegible] unacceptable."
286 LET string$, t$ = trim$(ucase$(string$))
287 IF len(string$) = 0 then CAUSE EXCEPTION 124, "Date
format string [illegible] & string$ & [illegible] is unacceptable."
288 LET [illegible] = " " ! initialize string with leading blank
289 LET delimiter$ = "" ! initialize delimiter holder
290 LET [illegible] = 1
291 FOR i=1 to [illegible]
292 SELECT CASE t$[i:i] ! get character of string
293 CASE "Y"
294 [illegible] = [illegible]
295 CALL [illegible]
296 LET [illegible] & "Y"
297 LET [illegible] = [illegible] the [illegible] of
[illegible]
298 CASE "M"
```



DJComputers.cz

```
299              [illegible]
300              [illegible]
301              LET d$ = d$ & [illegible]
302              LET length = counter   ! determine type of
year to return
303        CASE [illegible]
304              [illegible]
305              [illegible]
306              LET d$ = d$ & [illegible]
307              LET length = counter ! determine type of
year to return
308        CASE else ! assume its a delimiter character
309              LET delimiter$ = [illegible]  ! use only the last
delimiter found
310              LET i = i - 1 ! back up the
order_date_strings character
311              [illegible] character to allow for
delimiter
312        END SELECT
313     NEXT i
314     IF len(delimiter$) = 0 then LET delimit_flag = 0 else
LET delimit_flag = 1
315     IF length = 2 or length = 4 then ! length year OK
316        IF [illegible] then       ! length month OK
317           IF length [illegible] then
318              ! length day OK
319              LET [illegible] = string$
320              LET order_date_strings = d$
321              EXIT SUB
322           END IF
323        END IF
324     END IF
325     CAUSE EXCEPTION 124 "Date format string [illegible] & string$
& [illegible]"
326     SUB find_equal [illegible]
327        LET counter = 1 ! count [illegible] times Y, M, or D
occurs
328        FOR k=2 to len(t$)      ! find [illegible] t$
329           IF t$[k:k] = s$ then LET counter = counter + 1
[illegible]
330        NEXT k
331        LET [illegible] order_date_format$
string
332     END SUB          ! end of 'Find_Position'
333 END SUB             ! end Change_Date_Format

334 SUB parse_date(dat$,year,month,day)
335     ! EXCEPTIONS: 125, "Date [illegible]
delimiter$ & [illegible]"
336     !                 126, "Illegal date string."
337     LET year, month, day = 0
338     LET d$ = trim$(dat$)
339     SELECT CASE delimit_flag
340     CASE 0 ! no template delimiters used
341           WHEN error in
342              IF [illegible] then LET flag = 1
! month is numeric
343           USE
344              LET flag = 1
345              CALL [illegible]   ! [illegible] error number
346           END WHEN
347           IF flag = -1 then [illegible]
348              FOR i = 2 to 4     ! step through the order of
the template
349                 SELECT CASE order_date_strings$[i:i]
350                 CASE "Y"
351                    WHEN error in
352                       IF [illegible] then LET year =
val(trim$(d$[1:length]) else LET year = val(trim$(d$))
353                    USE
354                       CALL parse_date_error
355                       EXIT SUB
356                    END WHEN
357                    LET d$ = trim$(d$[length+1:[illegible]])
! trim off year
358                 CASE "M"
359                    WHEN error in
360                       IF [illegible] then LET month =
val(trim$(d$[1:length])) else LET month = val(trim$(d$))
361                    USE
362                       CALL parse_date_error
363                       EXIT SUB
364                    END WHEN
365                    LET d$ = trim$(d$[length+1:[illegible]])
! trim off month
366                 CASE "D"
367                    WHEN error in
368                       IF [illegible] then LET day =
val(trim$(d$[1:length])) else LET day = val(trim$(d$))
369                    USE
370                       CALL parse_date_error
371                       EXIT SUB
372                    END WHEN
373                    LET d$ = [illegible]
! trim off day
374                 END SELECT   ! of CASE
order_date_strings$[i:i]
375              NEXT i
376     CASE 1 ! delimiters used for the date
377           FOR i = 2 to 4 ! step through the order of
the template
```

**THIS IS NOT A COMPLETE LISTING**

Complete source code and listings can be found on the *AC's TECH* disk.

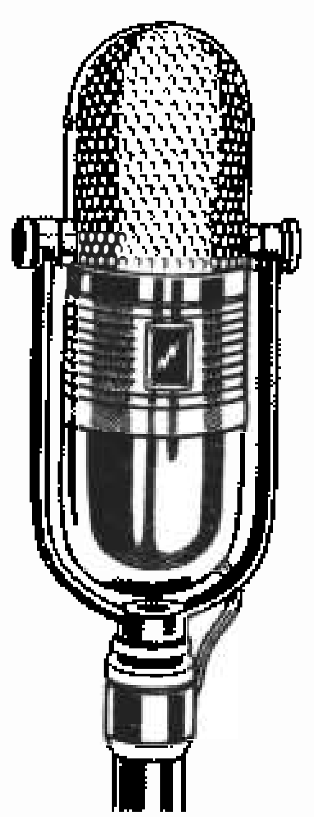
*Please write to:*
*T. Darrel Westbrook*
*c/o AC's TECH*
*P.O. Box 2140*
*Fall River, MA 02722*



DiConnuters.cz

Build Your Own

# SOUND



*by John Iovine*

DJComputers.cz

MOST OF US HAVE HEARD THE AMAZING sound capabilities of the Amiga computer. The ability to play digitized sound is one of the favorite features of the computer. To record your own sound you need an audio digitizer. The sound digitizer project described in this article allows the computer to sample sound. The sound sampler is compatible with most commercial software packages (QuasarSound, Audiomaster & others) as well as some PD software available on Fish disks (i.e. PerfectSound on Fred Fish #50).

The advantage of this project is that the electronics and components are kept to a bare bones minimum. This simplifies construction, lowers cost and improves the likelihood that you will actually build the project and that it will work successfully.

The heart of every digitizer is the ADC (Analog to Digital Converter) chip. This chip is responsible for reading an analog signal and outputting a binary number equivalent. In this case the analog signal to be sampled (digitized) is the audio input from a standard microphone. The binary number outputted by the ADC chip is read by the Amiga computer via its parallel port and stored in memory.

The ADC chip in this project is capable of digitizing 50,000 samples per second with an 8 bit (0-255) resolution.

## Sound Sampling

When recording, the ADC chip reads the voltage of the waveform at that particular instant and presents the binary number to the Amiga. The Amiga reads the number, stores it in memory, and signals the chip for the next sample. This continues for as long as sound is being recorded. The ADC chip follows and the computer records the basic shape of the original waveform.

During playback, the computer reads the binary numbers in sequence and outputs a proportional voltage via a sound channel. The output voltage varies in synchronization to the recorded signal, thereby playing back the digitized sound.

## Cycle Time

Sampling speed of the digitizer is important. It determines the fidelity and maximum frequency of the analog signal that the computer can record. Fortunately for us this has all been worked out long ago, its call the Nyquist criterion. It simply states that to digitally record an analog signal accurately you must sample at twice the maximum frequency of the analog signal. If you fail to meet this criterion you can not be sure of the accuracy (fidelity) of the digitized sound.

Our ADC chip can sample at 50,000 samples per second which exceeds the sampling speed of the sound software to date.

# DIGITIZER

## If you have not tried digitized sound on the Amiga, use this hardware project to make your Amiga "listen up."

Typically to record voice or simple sounds (i.e. bang, bell or tone) slow digitizing speed may be employed. More complex sounds like music and higher fidelity require faster sound digitizing. Sampling speed is determined by the software.

### Circuit Description

Look at the schematic illustrated in Figure 2. The circuit is easy to understand. The microphone input is fed into an 8-pin audio amplifier chip (LM386). The output from the audio amplifier IC is fed to the signal input on the ADC chip. The 8 bit number from the ADC connects to the Amiga 8-bit parallel (printer) port. The parallel port also supplies power to the audio digitizer circuit by lines 14 (+5V) and 22 (Ground).

Pins 2 through 9 on the parallel port are 8 bi-directional data lines. These pins are usually labeled DB0-DB7 in computerese. The Amiga computer reads the 8-bit binary number outputted from the ADC chip using these pins. Pin 14 supplies the +5 volts needed to power the project. Pin 22 is the ground. Pin 1 is a strobe pin that connects to the ADC's CE (chip enable) and RD (ready) pins.

The project is simple enough to build and wire without using a custom made PC board.

### Testing the Circuit

Testing the circuit depends upon which audio software you are using. Adjust the volume control until you have the appropriate recording level on your software.

If the circuit doesn't work, recheck your wiring against the schematic.

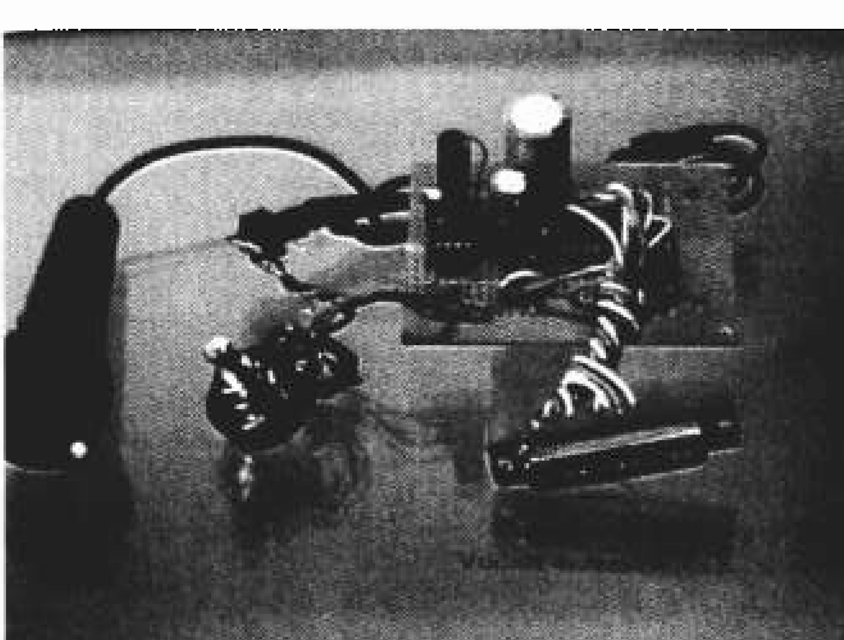
**Right: This "bare bones" project will allow you to produce sound files for other projects.**

### Voice Recognition

There is an interesting voice recognition program you can run using this digitizer. The magazine and accompanying program disk is available from PiM Publications, Inc. The magazine to order is *AC's TECH* Volume 2 Number 2. An updated version of the software is also available on the *AC's TECH* Volume 3 Number 4's accompanying disk.

*Caution: All projects are supplied on an "as is" basis. Although the author has built and tested this project for this article, neither the author, PiM Publications Inc. or its employees bear any responsibility for this project or its intended use.*

→



## Parts List

U1 Maxim 165 ACPN Chip
U2 LM-386 Audio Amp Chip
Q1 PN 2N2222 Transistor
R1 100K ohm 1/4 watt
R2,R4 4.7K ohm R3 16K
R5 1K
R6 10K Potentiometer
R7 220 ohm C1 100 pf
C2 4.7 uf
C3 220 uf
C4 10 uf
C5 100 uf
C6 1000 uf
Misc:
1/8" input jack
microphone
DB-25 Male connector
PC board
project case

U1 Maxim 165ACPN @$15.95 each are available from:
Images Company
POB 140742
Staten Island NY 10314
(718) 698 8305
add $5.00 Postage & Handling NYS residents add 8.25% sales tax
All other parts and components are available from your local Radio Shack.

Statement of Ownership, Management and Circulation
1A. Title of Publication: AC's Tech for the Commodore Amiga 1B. Publication No.: [illegible] 2. Date of Filing: 10/1/95 3. Frequency of Issue: Quarterly 3A. No. of Issues Published Annually: 4 3B. Annual Subscription Price: $44.95 US. 4. Complete Mailing Address of Known Office of Publication: P.O. Box 2140, Fall River, MA 02722-2140. 5. Complete Mailing Address of the Headquarters of General Business Offices of the Publisher: P.O. Box 2140, Fall River, MA 02722-2140. 6. Full Names and Complete Mailing Address of Publisher, Editor and Managing Editor: Publisher, Joyce A. Hicks P.O. Box 2140 Fall River, MA 02722-2140; Editor, Donald D. Hicks P.O. Box 2140 Fall River, MA 02722-2140. Managing Editor, Donald D. Hicks P.O. Box 2140 Fall River, MA 02722-2140. 7. Owner: PiM Publications, Inc. P.O. Box 2140 Fall River, MA 02722-2140, Joyce A. Hicks P.O. Box 2140 Fall River, MA 02722-2140. 8. Known Bondholders: None. 9. For Completion by Nonprofit Organizations Authorized to Mail at Special Rates: Not Applicable. 10. Extent and Nature of Circulation: (X) Average No. Copies Each Issue During Preceding 12 Months: (Y) Actual No. Copies of Single Issue Published Nearest to Filing Date. 10A. Total No. Copies (X) [illegible] (Y) 6,582. 10B. Paid and/or Requested Circulation 1. Sales through dealers and carriers, street vendors and counter sales (X) [illegible] (Y) [illegible] 2. Mail Subscription (X) 1,573 (Y) [illegible] 10C. Total Paid and/or Requested Circulation (X) 5,751 (Y) 4,756. 10D. Free Distribution by Mail, Carrier or other Means Samples, Complimentary, and other Free Copies (X) 0 (Y) 0. 10E. Total Distribution (X) [illegible] (Y) 4,756. 10F. Copies Not Distributed: 1. Office Use, Left over, Unaccounted, Spoiled after Printing (X) 2,024 (Y) 1,632. 2. Return from News Agents (X) [illegible] (Y) [illegible] 10G. Total: (X) [illegible] (Y) 6,582.

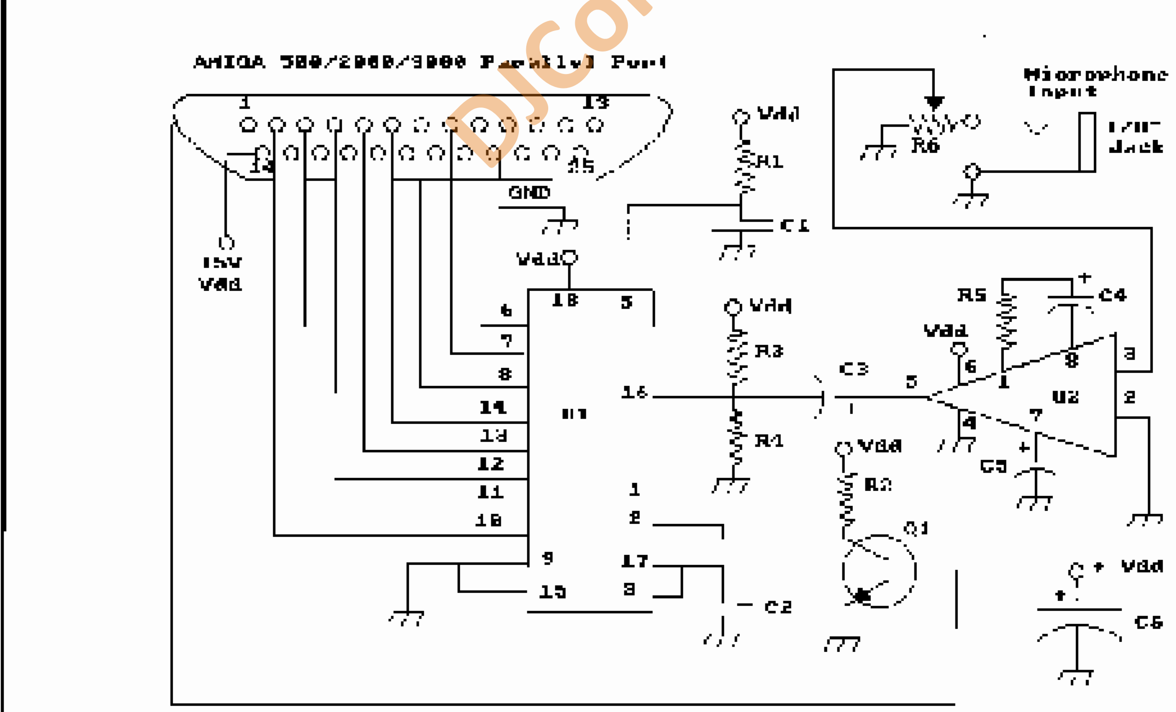



Figure 2

Technical Writers
Hardware Technicians
Programmers
Amiga Enthusiasts

Do you work your Amiga to its limits? Do you do create your own programs and utilities? Are you a master of any of the programming languages available for the Amiga? Do you often find yourself reworking a piece of hardware or software to your own specifications?

If you answered yes to any of those questions, then you belong writing for *AC's TECH*!

*AC's TECH for the Commodore Amiga* is the only Amiga-based technical magazine available! We are constantly looking for new authors and fresh ideas to complement the magazine as it grows in a rapidly expanding technical market.

Share your ideas, your knowledge, and your creations with the rest of the Amiga technical community—become an *AC's TECH* author.

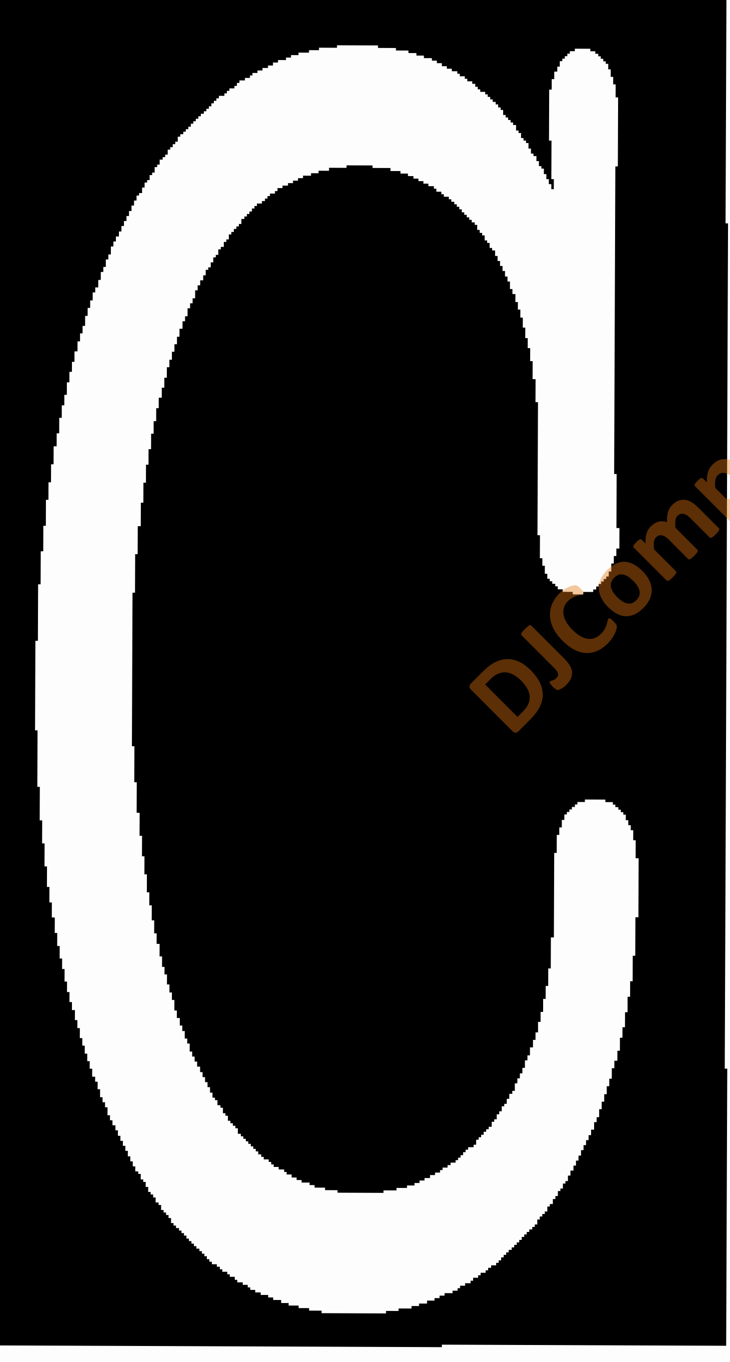
For more information, call or write:

AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

1-800-345-3360

z3.slaufpdumo.pl

# A better way to



When most people think of C++, they think of the support it provides for Object-Oriented Programming (OOP). They forget that C++ is an extension of C and can, with some effort, be used instead of C. Advantages to a C programmer in using C++ include stronger type checking, more consistent treatment of user defined and built-in types, and some powerful extensions to C. The purpose of this article is to demonstrate how C++ can be used as a better version of C, and to show that C++ is useful in a C programming environment without the need to learn the OO paradigms that are supported by C++. I do not mean to imply by this that Object-Oriented Programming is in some way inferior, or even that the way that C++ implements OO concepts is incorrect. I just feel that at this time, with the lack of C++ tools in the Amiga community and the heavy emphasis on C, that this method is the most logical way to start using C++.

by Paul Gittings

DJComputers.cz

# A variety of options await the C programmer

I would even go so far as to recommend that all C programmers get a C++ compiler and start using it to compile their C code. For a start, C++ has stronger type checking than C and this will require extra discipline on the part of the programmer, but s/he will be rewarded with wasting less time finding irritating bugs. C++ also has some simple extensions to standard C programming concepts. Hopefully I will convince readers that using C++ as a better C is a useful way to introduce oneself to C++. Also, I hope that having mastered these extensions that you will go on and explore the full power and benefits of all the features in C++.

C Compilers for the Amiga, with the exception of some of the public domain compilers, are ANSI C compatible. Since most C programmers tend to be more familiar with K&R C I will explain new C++ features relative to K&R C. Programmers already familiar with ANSI C will have encountered some of the features discussed in this article. But be warned, there are subtle differences between the way ANSI C and C++ implement some of them.

The reference I used for K&R C is "The C Programming Language", first edition, by Brian W. Kernighan and Dennis Ritchie. For ANSI C I used the following three books: "The C Programming Language Guide", second edition, by Brian Kernighan and Dennis Ritchie; "The Waite Group's Essential Guide to ANSI C" by Naba Barkakati; and "Standard C" by P. Plauger and Jim Brodie. During the writing of this article I used two ANSI C compilers to verify the operation of ANSI C: a registered version of Matt Dillon's DICE C compiler (version 2.06.19, with the DICE pre-processor supplied with Comeau C++), and Markus Wild's port of version 2.3.3 of the Free Software Foundation's gcc compiler (see sidebar article "Swiss Army Knife Compiler") using the "-ansi" option (this option disables many extensions in the gcc compiler and results in a closer conformance to the ANSI C standard).

The reference I used for C++ is "The Annotated C++ Reference Manual" by Margaret Ellis and Bjarne Stroustrup, May 1992 (I will refer to this book as ARM). I currently use both Comeau C++ (with DICE as a back end, see sidebar article) and Markus Wild's port of version 2.3.3 of g++. I have come across places where g++ does not conform to the C++ language as defined in the above book; I will identify such discrepancies.

The first new feature of C++ (which is also supported by DICE) we will look at is very simple to understand and use. It is a new comment style. This is a comment to end of line delimiter, //. Anything that appears after the // up until the end of line is treated as a comment.

```
int count;      // counter of items to test
// #ifdef Debug_Delay
```

A simple feature, but one that I find most useful.

Arguably the most important extension of ANSI C over K&R C is function prototyping. A function prototype is used to specify the number and type of arguments required by a function. Function prototypes can be used in both function declarations and function definitions. Without the information supplied by a function prototype a compiler cannot make a decision on whether a function call has the correct number of arguments or if they are the correct type. No such capability exists in K&R C. So a K&R C compiler cannot know if a call to a function has the correct number or type of arguments. Consider the following example of a function declaration in K&R C:

```
        extern long dosomething();   /* declare function */
        main()        {              long result;            result =
dosomething( 1, 2, 6, "Stuff");       }
```

The K&R compiler would not know how many arguments dosomething expects, so the compiler could make no decision as to whether the call specifies the correct number and type of arguments. In K&R C the only thing that function declarations can be used for is to specify that a function returns something other than an int; in the above example the declaration of the dosomething function specified that the function returned a long. There is no way K&R C, by the use of function declarations, can inform the compiler of the correct type or number of arguments that a function is expecting. This changed in C++ with the addition of function prototyping and a watered down version was also later included in the ANSI C standard.

In C++ and ANSI C a declaration can now be used to specify the number and type of arguments. Declarations of this type are used by the compiler to check that function calls have the required number of arguments and are of the correct type. A function declaration in C++ and ANSI C for the dosomething function above might look like:

```
        extern long dosomething(int one, int two, int point_count, int
length, char *string);
```

As you can see the declaration states the type of each of the arguments and a name for each argument is supplied. The argument names used in the declaration do not need to be identical to the names used in the actual function definition, only the types need to match. In fact the names can be left out of a function declaration:

```
extern long dosomething(int, int, int, char*);
```

For readability it is usually a good idea to leave the names in. In ANSI C and C++ the function prototyping format is also used in the function definitions themselves. For example the main function is

usually defined as:

```
int main(int argc, char *argv[]) [illegible]
where as in K&R C the usual definition is,
int main(argc, argv) int argc; char *argv[];
[illegible]
```

Undeclared functions in ANSI C are treated as they would be in K&R C, that is, the function is assumed to have a return type of int and have any number of arguments. C++ is stricter than this, and a function can be called only if it has already been declared or defined in the scope of the call. The following file would compile successfully under ANSI C and K&R C but would result in a compile time error when compiled with a C++ compiler:

```
int function( ) [illegible] int result;
result = unknownFunc(); /* undeclared function */
return result;
```

g++ is not as strict as it should be and it will compile the above treating unknownFunc as a function which returns an int. It will, however, generate a warning which states that unknownFunc is an undeclared function.

C++, unlike ANSI C, also requires that a function declaration must contain a function prototype. That is, all the argument types must be specified in the function declaration. By making this mandatory, a C++ compiler is always assured of being able to check function calls for the correct number and type of arguments.

This is all well and good but suppose you want to write a function like printf which has a variable number of arguments.

In K&R C (and in ANSI C) to declare such a function is easy since the compiler assumes an unknown number of arguments anyway. Since C++ now forces you to declare the number and type of such arguments how can you write a function with a variable number of arguments? Well, both C++ and ANSI C have a standard way of declaring a function with a variable number of arguments. Such definitions use ellipses, "...", to indicate zero or more additional variables of an unknown type. For example

```
int my_printf(char *fmt, ...);
```

Defines a function with one or more arguments. Note, there should be at least one argument specified before the ellipses. The leading argument is required by a set of macros used to access the trailing arguments.

Accessing a variable number of arguments in the early days of C programming required machine specific tricks which were nearly always unportable between different machines. I once spent the better part of two days trying to figure out why a compiler, which worked on a 68020 based machine, would not work when recompiled on a Sun SPARCStation. The problem was due to the tricks used to access a variable number of arguments in a function. To get around this portability problem, ANSI C defines a set of macros that can be used to access a variable number of arguments. The macros are defined in the header file stdarg.h. The C++ Reference Manual also recommends that these macros be used when accessing a variable number of arguments from within a C++ function. For these macros to work

## Using Comeau C++ 3.0 with DICE

Comeau C++ 3.0 for the Amiga officially supports SAS/C version 5.10a and above as well as Manx Aztec C version 5.0d and above. The manual states that other C compilers should be able to work as back ends to Comeau C++. The manual also warns that Comeau Computing will not guarantee that Comeau C++ will work with any but the supported compilers. I took a bit of a gamble when I ordered Comeau C++ since I do not own one of the supported C compilers. I do however own a registered version of Matt Dillon's DICE C compiler (version 2.06.19).

I was actually very surprised at how easy it was to incorporate DICE into Comeau C++. I followed the standard procedure of installing Comeau C++ onto my hard drive. However, when the install script asked me which of the supported compilers I used, I respond no to each of the compilers listed. After the install script had finished, everything had been copied from the distribution disks to my hard drive except for the header files.

Part of the installation procedure is to create a set of C++ header files based on the C header files used by the C compiler on the system. Since the install script does not support DICE I had to create the C++ header files myself. I first created a directory into which the C++ header files would be stored, called dinclude (this directory was created in the same directory as the rest of the Comeau directories on my hard drive). Next I used the supplied utility c30/include to convert my DICE and Commodore header files.

This program takes two arguments; the first argument is the name of the directory in which you want to store the C++ files, the second is where the C files can currently be found. On my system I used;

```
c30/include dinclude DINCLUDE:
```

This worked up to a point. On my system there is a symbolic link in the DINCLUDE: directory to where I keep the Commodore AmigaDOS include files; apps:cc/2.0/include. The include utility seemed to have a problem with this symbolic link and terminated before converting all the files. To get around this problem I removed the symbolic link from the DINCLUDE: directory and then repeated the conversion process in two steps:

```
c30/include dinclude DINCLUDE: [illegible] dinclude/amiga20 c30/include dinclude/amiga20 apps:cc/2.0/include
```

This seems to have worked. However, as the include utility seems to always print out what appears to be an error when it finishes processing, I am not a hundred percent sure if everything is correct.

The next step was to copy the standard Comeau C++ headers from the first installation disk onto my hard drive. This was easily achieved with;

```
copy df0:include/#?.h dinclude
```

As part of the installation the install script creates a file called forS, into which it puts various assigns; it is intended that this file should be added

DJcomputers.cz

however, there must be at least one fixed argument. Good examples of using stdarg.h can be found in the second edition of "The C Programming Language" pages 155-156 and the ARM pages 146-148. The use of these macros is the only guaranteed way to write portable code to access a variable number of arguments.

It is possible in C++ to declare a function with an unknown number of arguments and no leading arguments,

```
int oldc(...);
```

which states that oldc has any number of arguments including zero. Since there is no fixed argument, the stdarg.h macros cannot be used. Thus there is no guaranteed way to access the arguments to function oldc. This was included in C++ to provide an equivalent to K&R C's unchecked function call, and has the same portability problems. Don't use it!

In C++ and ANSI C you can also explicitly tell the compiler there are no arguments to a function by use of the void type. The following declares a function which has no arguments:

```
int noargs(void);
```

In C++ the following also declares a function with no arguments:

```
int noargs();
```

whereas in ANSI C it would be interpreted as a declaration of a function with any number of arguments.

Void is a new type in C++ and ANSI C. As well as its use in the above declaration, it can also be used to indicate that a function does not return anything:

```
extern void noreturnvalue(void);
void
noreturnvalue(void)
{
        // no value
        return;
}
```

Also, in C++ and ANSI C a pointer to a void should be used as the generic pointer. This is a role that was fulfilled by char* in K&R C. For instance in K&R C the function malloc was declared to return char*; in C++ and ANSI C it is declared to return void*.

While both C++ and ANSI C support the void type there is however a difference in their treatment of void pointers. In ANSI C a pointer to a void can be assigned to a pointer of any other type without a cast:

```
char *cptr;
cptr = malloc(sizeof(char)); // malloc returns void*
```

In C++, however, this is an error; you have to explicitly cast the void pointer to the appropriate type:

```
cptr = (char *) malloc(sizeof(char));
```

Again, g++ differs to the "C++ Reference Manual" as it will accept such assignments without a cast; it does, however, produce a warning.

to the scuser-startup file. I had to change the assign for CC30include: in this file to point to the dinclude directory.

```
assign  CC30include:  CC:comeau/dinclude
```

Part of the installation process also installs a version of Matt Dillon's C pre-processor, dcpp, into the c30 directory. I checked the version number of this copy of dcpp (dcpp -v) and since it was newer than the version I currently had installed on my system I moved it from c30 to DCC:bin. I then had to change another assign in the file forS:

```
assign  DICE:  DCC:bin
```

At this point I decided to actually try out Comeau C++. I did not really expect anything to work since I thought I would still have to change the ARexx script, ARexx30/como.rexx, to work with DICE. First I executed forS to set up the assigns required by Comeau C++. Then I tried to compile one of the test programs supplied with Comeau C++, rx ARexx30/como.rexx -V cctest.c. I was a bit surprised by the result. While the compile did not work, the como script had identified the fact that I used DICE and had used the DICE compiler with what, at first glance, looked like the correct arguments. The compile failed because the DICE linker, dlink, could not find the link libraries specified by the como script. On my system the DICE standard link libraries are called c.lib and amiga20.lib. The como script, however, was using cl.lib and amigal20.lib. This was easily fixed by editing the como.rexx file and making the appropriate changes (I also changed the ARexx/c.rexx file but I'm not sure what this is used for). There are two spots in como.rexx from which a call to dlink is made, and the library names in each of them had to be changed.

I copied the modified como.rexx script file to REXX: where I keep all my ARexx scripts. I added the file forS to my systems scuser-startup file. Then I added the following alias to my systems sshell-startup file:

```
alias  como  rx REXX:como.rexx
```

I then rebooted my machine and Comeau C++ has worked fine on my system ever since (with the one caveat that I cannot use structure return types since the version of DICE that I use does not support them). I have not had to make any changes since I made the above few. I may however do a few modifications to the como.rexx file to get it to print out fewer messages and to hard code the fact I use DICE into it (it currently scans the system list of assigns to figure out which compiler is installed). Also I may change the como.rexx script to use the +a1 flag as a default option rather than +a0; this causes the Comeau C++ compiler to output ANSI function declarations rather than K&R style declarations, DICE seems to work better with the ANSI declarations.

So you can see that configuring Comeau C++ to work with DICE is a fairly easy procedure. But I must again repeat that DICE is not supported by Comeau Computing as a back end to Comeau C++ and neither they, nor I, can guarantee that you won't have any problems. However, in my current use of Comeau C++ I have not encountered any problems with this combination.

—PG

message.

Since I have used malloc in the above example I should point out that C++ has two new operators which replace malloc and its companion function free, called new and delete. new is used to allocate memory and delete is used to free it up again.

```
int *range;
    // declare pointer          ...     range = new int;
  // get some storage           ...     delete range;
    // free the storage
```

In the above fragment note that the new operator accepts the name of the type that you are trying to allocate space for as an operand. You do not need to cast the result of the new operator as it returns a pointer of the correct type (i.e. pointer to the operand type). Unlike malloc you do not have to specify how big a space to allocate for simple types or structures. However, as with malloc, you should check that the allocation of space was successful by checking that the pointer returned is not equal to zero. Also, the space allocated by new is not guaranteed to be initialized, but this can be accomplished for simple types by using a slightly different syntax:

```
        int *p;
        p = new int(0);  // allocate and initialize to 0
```

The allocation and deallocation of arrays also requires a different syntax:

```
        char *buf;  // declare pointer          ...     buf =
new char[50];  // allocate array of 50 char     ...     delete []
buf;    // deallocate array
```

Note that when deallocating an array you have to inform the delete operator of this by specifying [] before the pointer name. In older C++ compilers you also had to put in the size of the array, but this is no longer required. You cannot use new to initialize an array, or structure, as is possible when allocating space for simple types.

There are a number of things to be aware of when using the delete operator. If the pointer you are deleting is not zero, it must be a pointer that was returned by new. The result of calling delete on a pointer not obtained from the new operator is undefined in C++ and the outcome will usually be harmful. Consider:

```
        void bad()      {       char *buf;      char
*buf2 = buf;            char *buf2;
            [illegible]     // [illegible]
buf = new char[50];             buf2++;         // decrement pointer returned
by new          delete [] buf2; // error!
                return;         }
```

In addition the result of deleting an array without specifying the [] is undefined, as is deleting an individual item with the delete [] syntax. It is unlikely that a C++ compiler will always be able to detect the occurrence of all such errors. It is safe to assume that only damage will result from such operations and it is wise to do everything possible to avoid such situations.

It is possible to delete a pointer with the value zero, this is guaranteed to be harmless. Thus, deleting allocated space does not require checks to see if the item being deleted were in fact successfully allocated by new. This can simplify error recovery code.

```
        void good()
        {               char *p = new char[50];
```

# GCC: The "Swiss-Army Knife" Compiler

In the world of Unix the GCC compiler is one of the best known compilers around. One reason for this is that it is excellent value for money. In the first place it is not one but three compilers; a C compiler (both K&R and ANSI standards are supported via command line options), a C++ compiler and an Objective-C compiler. It is also freely distributable. Who would produce such an item for free?

GCC is a product, if you can call it that, of the Free Software Foundation (FSF). The FSF is a collection of programmers who are trying to combat software hoarding (ie copyright) of large software houses by writing quality software and making it freely distributable. While freely distributable they are not in the Public Domain; they are instead covered by the GNU PUBLIC LICENSE which some people refer to as the "copyleft". As a recipient of a program covered by the "copyleft" you are free to change it, redistribute it, even sell it. You are however under certain obligations, these include;

a) informing anybody receiving your program of their rights under the "copyleft"
b) in the case of modification, make it clear that you have modified the program.
c) make the source code of your program available to people who use it.

The GNU in the name of this license stands for "Gnu is Not Unix". The GNU project is one of the major projects of the FSF, the aim of which is to create a freely redistributable operating system which is compatible with Unix. Since you need a compiler to write an OS, one of the first programs to make an appearance out of the GNU project was the GNU C compiler, gcc. Each new release of gcc has seen it grow better and more powerful.

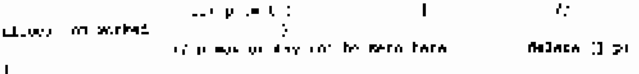
Version 2.0 included not only the C compiler but also incorporated g++ (which is the GNU C++ compiler) and an Objective-C compiler. Due to a mammoth piece of work, Markus Wild has ported several version of GCC to the Amiga; the lastest version I am of aware of is version 2.3.3.

Markus not only ported the compiler (the core C compiler and the C++ and the Objective-C compilers) but also numerous support tools (assembler, linker, archive maintainer, libg++ etc). He also wrote a library to emulate many functions available in Unix libraries. As required by the copyleft, Markus included the source to all his changes in the form of diff files; they list the differences between the FSF release and Markus' version; another program called patch can be used to apply the diffs to the FSF files to produce the Amiga specific version. NOTE: neither diff or patch are supplied with Markus' distribution of GCC, however both are available on various Fish Disks.

To give you some idea of what you get in the GCC distribution here is a list of the major components:

1) GCC 2.3.3. This compiler will compile C (both ANSI and K&R), C++ and Objective-C (there is a problem with Objective-C however).

DJComputers.cz



One final warning: on delete, the value of the pointer after a delete operation may or may not be changed. Don't assume that it is set to zero and never use a pointer after it is deleted.

The new and delete operators can of course be used to allocate and deallocate structures:

```
struct limits {          int low;
int high;      };
void main(void) {               limits *limits_p = new
limits;         ....
      delete limits_p;         }
```

In the above, the call to new is made when initialising the pointer limits_p. Also notice that limits_p was declared to be a pointer to limits and not as a pointer to struct limits as it would in ANSI C. This is because C++ automatically creates a new type for any tagged structure. This does away with one of the common uses in C of the typedef statement.

C++ and ANSI C both allow structures to be passed as arguments to, or returned as a result from, a function (GNU C, at least the version I have, does not support structure return types). In K&R C this was not allowed and a pointer to a structure would have to be passed instead. Passing entire structures can be inefficient since, when a variable is passed to a function, the function effectively has its own private copy of the structure and this copying of large structures can incur a heavy price. An advantage of passing a structure, or any variable for that matter, rather than a pointer to it is that any changes the function makes to the structure it makes to its own private copy; the changes are not made to the structure in the calling program. This type of argument passing is called "pass by value" and is the only way that K&R and ANSI C allow arguments to be passed to a function. This is why, in K&R and ANSI C, if a variable in a calling program is to be changed by a function call then a pointer to the variable has to be passed to the function. This places the onus on the person writing the function calls to ensure that a pointer, rather than a variable, is passed. To place the responsibility for such decisions back where it should be, C++ introduces another method of argument passing, "pass by reference".

When an argument is passed by reference no local copy is made. In fact the local variable is a reference to (same as) the variable in the calling program. Changing the value of the local variable will change the value of the variable in the calling function.

To indicate that an argument is to be passed by reference the & operator is used in function declarations and definitions. The use of call by reference arguments removes the need for passing the address of variables in function calls and explicit pointer dereferencing inside the function. The following is a very simple example of a function which accepts one argument that is passed by reference.

```
#include <stdio.h>

// increment the argument passed by reference     void
increment(int &i)    {        i++;        return;     }
int main()      {        int my_value = 0;
   increment(my_value);        printf("the value is %d\n",
my_value);
   return 0;      }
```

2) gas. The GNU assembler.
3) gld. The GNU linker, this links Unix style object files not Amiga object files.
4) hunk2gcc. A program written by Markus Wild to convert Amiga link libraries (eg. amiga.lib) into a form that gld can handle.
5) ar and ranlib. Programs to manage Unix style link libraries.
6) libg++. The GNU C++ library, containing many useful classes and libraries including a streams library.
7) ixemul.library. An Amiga shared library written by Markus which contains many of the standard Unix functions. There are also various link libraries included to allow access to this library.
8) Unix style man pages are supplied for most programs and ixemul.library routines. A program called "man" is also supplied to read them.
9) various header files ( C and C++ ). Does not include Commodore Header files (which have to be obtained from Commodore).
10) Various documentation in the form of info files, and infoview, a program to browse this online information.

There are however a number of things to be aware of with this release of gcc 2.3.3:

1) There are no Objective-C includes; therefore as far as I can tell there is no way to use it since all the Objective-C programs I have seen start with # import <objc/Object.h>
2) The only information supplied on g++ is a man page.
3) There is no documentation on the Objective-C compiler.
4) The object files produced by gcc/g++ are not the same as Amiga object files; ie, you can't use an Amiga linker to link them. Also the GCC linker gld does not understand the format of Amiga object files (or libraries). To help get around this problem Markus has included a utility called hunk2gcc, which will take an Amiga object file and convert it to a form which can be used with GCC's Unix style linker.

Due to the size of this distribution you are unlikely to see this compiler on a Fred Fish disk. There was some talk awhile back on the Internet about this compiler being included in a CD-ROM containing various non-commercial Amiga programs (PD, Freeware, Shareware etc), but I do not know if the CD-ROM was ever released. It is, however, freely available on the Internet; check the ftp site amiga.physik.unizh.ch. To install and run this compiler you will need; a hard drive with at least 8MB of disk space free, about 5MB of memory, and AmigaDOS 2.04 or later is recommended.

Markus has done a first rate job on porting GCC and related tools to the Amiga. If you need a low cost C or C++ compiler and you are willing to put up with the problems of the strange (to Amiga eyes anyway) format of the object files, Markus' port of GCC could just be what you need. —PG

Workbench Window Icons Tools

Backdrop

Execute Command

Amazing Computing

# What's the best way to improve productivity on your Workbench?

DJComputers.cz

# With
# *Amazing Computing*

*Amazing Computing for the Commodore Amiga*, *AC's GUIDE* and *AC's TECH* provide you with the most comprehensive coverage of the Amiga. Coverage you would expect from the longest running monthly Amiga publication.

The pages of *Amazing Computing* bring you insights into the world of the Commodore Amiga. You'll find comprehensive reviews of Amiga products, complete coverage of all the major Amiga trade shows, and hints, tips, and tutorials on a variety of Amiga subjects such as desktop publishing, video, programming, and hardware. You'll also find a listing of the latest Fred Fish disks, monthly columns on using the CLI and working with ARexx, and you can keep up to date with new releases in "New Products and Other Neat Stuff."

*AC's GUIDE to the Commodore Amiga* is an indispensable catalog of all the hardware, software, public domain collection, services, and information available for the Amiga. This amazing book lists over 3500 products and is updated every six months!

*AC's TECH for the Commodore Amiga* provides the Amiga user with valuable insights into the inner workings of the Amiga. In-depth articles on programming and hardware enhancement are designed to help the user gain the knowledge he needs to get the most out of his machine.

# Call 1-800-345-3360

will produce the output:

```
my_value = 1
```

In general you should declare arguments to be pass by reference when you wish changes made to the variable in the function to be propagated back. On the other hand, when you don't want such changes propagated back you should use the standard C method of argument passing, pass by value.

As stated earlier, pass by value can be an inefficient way to pass large structures. If the function requires read only access to the structure it might be better to declare the argument as pass by reference and also specify, with the const type specifier, that it is not to be modified. const is new to C++ and ANSI C; it indicates to the compiler that a variable is to be treated as if it is read only. Any attempt to modify a variable specified as const will generate a compile time error. In the following C++ example, the routine print_limits demonstrates how to use a combination of const and pass by reference as a more efficient version of pass by value.

```
[illegible]
struct limits { int min; int max; };
void print_limits( const limits &range) { printf(
"max = %d, min = %d\n", range.max, range.min);
}
int main() { limits my_limits;
my_limits.max = 10; my_limits.min = 1;
print_limits(my_limits);
return(0); }
```

If an attempt were made in print_limits to modify the values of range a compile time error would occur indicating that an attempt had been made to modify a read only value.

Another use of const is as a replacement of pre-processor macros used to define constant values. The following macro:

```
#define MAX_BUFFER (20)
could be replaced by
const int MAX_BUFFER = 20;
```

A const without a type specifier is assumed to be an int so the above could have been written as

```
const MAX_BUFFER = 20;
```

This use of const variables has a number of advantages over macros; they can be type checked by the compiler and the name of the variable will show up in a symbolic debugger and, of course, the usual scoping rules apply to such variables. In ANSI C however, such const variables cannot be used as the size of arrays. For example the following will not compile under ANSI C:

```
const int MAX_BUFFER = 20;        char buffer[MAX_BUFFER];
```

As far as ANSI C is concerned MAX_BUFFER is still a variable! In C++, however, the above will compile correctly.

Another feature of C++ which is sometimes used instead of macro constants is enumerated types. An example of an enumerated type in C++ and ANSI C:

## A Brief History of C & C++

C++ is first and foremost an extension to C. A C++ compiler will compile many C programs without any changes having to be made, and it will compile many more with just minor changes. The reason for this is that C++ has evolved from C and some thought was given to backwards compatibility during the design of the C++ language.

The C programming language in one form or another has been around since the late seventies. The C language was originally designed and implemented by Dennis Ritchie on a DEC PDP-11 at AT&T's Bell Laboratories. About 1978 the "standard" reference for this language was published; "The C programming Language", first edition, by Brian Kernighan and Dennis Ritchie. This version of the C language became known as "K&R" C, after the authors, and later in some quarters as "Classic" C (this name was probably inspired by a cola advertising campaign).

C was then implemented on a large number of machines under various operating systems. It was found that the original C standard was ambiguous in places and even incomplete in some areas (for instance it did not specify the C library). This resulted in many compiler implementors using differing interpretations of the standard. In addition, a number of new features were being added to some implementations. The result, a whole swarm of various C dialects with many incompatibilities between them. Many of the new features been added had being developed in another AT&T language project.

In 1980 Bjarne Stroustrup added classes, function argument type checking and several other features to K&R C; the resulting language was called "C with Classes". This was a fore runner of C++. Stroustrup continued with the development of "C with Classes" until about 1983 when it was redesigned, extended, reimplemented and renamed as C++. C++, after a few refinements, became generally available in 1985. C++ is still evolving and being refined. While an ANSI (ANSI X3J16) and ISO (ISO WG21) committee have jointly been working on a C++ language standard since 1991 (ANSI has been working on a standard since 1989) a standard has not yet been finalized. The current pseudo standard for C++ is the book "The Annotated C++ Reference Manual" by Margaret Ellis and Bjarne Stroustrup; this book is usually referred to in most C++ literature as the ARM. This book is also the base document for the ANSI C++ standardization effort; chapter 19 lists the resolutions made by ANSI/ISO standardization committee so far. These resolutions aid compiler implementors in their implementation of various language features so that when the standard is finalized their implementations stand a better chance of conforming to that standard.

Currently when C++ compiler implementors say that their compiler conforms to version 3.0 of C++ they are usually referring to the C++ language as implemented by Release 3.0 of AT&T's C++ to C translator, cfront, which in turn is based on the current content of the ARM. Comeau C++ 3.0 With Templates for the Amiga is a licensed port of the

DJcomputers.cz

```
enum days { Monday, Tuesday, Wednesday, Thursday,
Friday, Saturday, Sunday };
```

The enumeration constants Monday - Sunday are assigned, by the compiler, integer values starting at 0. It is possible to determine specific values for the enumeration constants and they do not have to be unique. The following is valid in both ANSI C and C++:

```
enum Bo_Type { Label1, part1 = 1,
Label2, ab 3, abt 3 };
```

The enumeration constants can be used in much the same way as macro constants are usually used. You can of course declare variables of enumerated types.

A variable of type enum days can be declared in both ANSI C and C++ with

```
enum days a_day;
```

Just as with struct, C++ automatically creates a type when it encounters an enum. So the above declaration can be written in C++ (and this is the preferred way) as:

```
days a_day;
```

An enumerated type can, as can struct types, be used in cast operations

```
a_day = (enum days) 1;      /* C++ or ANSI C */      a_day
= (days) 1;      // C++ only
```



It is of course possible in ANSI C to create an enumerated type with the typedef statement and achieve similar results to C++. If days had been declared as follows, the enum could also be dropped in ANSI C variable declarations and casts:

```
typedef enum { Monday, Tuesday, Wednesday, Thursday,
Friday, Saturday, Sunday } days;
```

The treatment of enumerated types differs somewhat between C++ and ANSI C. As stated earlier, when a compiler of either type encounters a definition of an enum type it assigns an integer value to each enumeration constant (Monday through Sunday above), starting at the left with 0. Since any enumerated type is a sub-type of int it is possible to assign an enum value to an int variable. So in ANSI C and C++ the following is legal, since Monday will be promoted to the int value of 0:

```
int i = Monday;
```

However, ANSI C and C++ will treat the following differently:

```
a_day = 5;
```

This is valid in ANSI C but not in C++. The above will generate a "type mismatch" error in C++ (however g++ seems to treat such a statement identically to ANSI C). In C++ a distinct integer type is created for each enumeration type; variables of one type cannot be assigned values directly from another type; variables of type days can only be assigned values of type days, ie Monday through Sunday, or from other variables of type days. This is another example of C++'s stricter type checking over ANSI C. To perform such an assignment, if you really need to, you would have to cast the value to the appropriate type.

C++ even has an alternative to the standard cast operator called explicit type conversion. This new operator looks like a function call and takes the form:

```
simple_type( value to be converted )
```

In the following example a float value is cast to an int value:

```
int i;      i = int(3.0);      // cast float constant to an
int
```

Explicit type conversion also works for simple types you create. Thus to assign an integer value to a variable of our enumerated type days any of the following could be used:

```
a_day = (enum days) 0;    /* ANSI or C++ */      a_day =
(days) 0;      // C++ only      a_day = days(0);      // C++ only
```

Since structures are not considered simple types, explicit type conversion in C++ will not work with structures. Thus the following line will produce an error:

```
mdays = limits(0,0);
```

In fact it will produce a rather cryptic error message:

```
"Cannot find a constructor"
```

Explanation of this error message requires that I own up to a little white lie I told above. Explicit type conversion for structures can in fact be made to work in C++. This requires the use of a new and powerful feature in C++ called a class, which can be thought of as an extension of a struct; actually in C++ a struct is just another name for class. In a class you can define functions, called member functions, as well as data fields. If limits had been declared with an appropriate member function, called a constructor, the above type conversion would have worked. The lack of such a function cause the above error message. An explanation of classes and constructors is beyond the scope of this article, but if there is enough interest an explanation of classes may form part of a future article.

C++ has yet another feature which can also be used as a replacement for some macros: the inline function specifier. This indicates to the compiler that a function so specified be considered for inlining; that is, when a call to the function is encountered, rather than generating code to actually perform the call the compiler will insert a slightly modified version of the function in place of the call. The compiler may, however, choose not to perform the inlining due to the outcome of some (inbuilt) heuristics. Also most C++ compilers have an option to turn inlining off (+d in Comeau C++); this option is useful when you want to debug a program. When is inlining useful? Suitable candidates for inlining are small functions of a few lines. The benefit is speed, inlined functions are executed faster and in some cases even memory is saved.

As you have probably guessed, inlined functions can be used instead of macros. Consider:

```
#define abs(a) ( (a) [illegible] )
```

which can be replaced with the inline function:

```
inline int abs( int a) { return a [illegible] }
```

Inline functions have a number of advantages over macro functions. They use the standard function syntax and, since the types of the returned value and arguments are specified, the compiler can type check calls to the function.

A further addition to C++ functions is that function names can be overloaded. This means that more than one function can have the same name as long as they can be distinguished by argument types. Consider the following:

```
int   IntTotal = 0;
float FloatTotal = 0.0;

void addToTotal( int x ) { [illegible] = x;
}
void addToTotal( float x ) { [illegible] +=
x; }
```

The above C++ code defines two different functions called addToTotal, one of which accepts an int value, the other a float. As long as the compiler can distinguish between all instances of the function by the number and type of their arguments any number of definitions can be given. You could, if you were perverse, have all your functions called g!. Needless to say, such an occurrence is not what this feature was designed for. An example of a better use would be in a sort library: rather than have functions called such things as sortString, sortInt, sortFloat etc. you could have all the functions called sort. The compiler would be able to identify which routine should be called by the type of arguments given in a function call.

The more technically inclined readers may wonder how a C++ compiler keeps track of the different versions of the function and passes this information to the linker, especially across compilation units. C++ encodes all functions names. This encoding, depending on the function name and the type of each of its arguments, is referred to as name mangling. In this way a unique identifier is generated by the compiler for each of the overloaded functions. These encodings also allow for type safe linkage across compilation units. Since the function names that the linker will see actually includes information on the function arguments, it guarantees that function calls will only be linked to a function if the arguments in the function call and the function are of the same type (ie, they have the same name). You can actually see these encodings in assembler output from a C++ compiler or via a symbolic debugger. Currently there is no debugger available for the Amiga which is C++ aware, therefore some knowledge of the encoding scheme will be required so that you will be able to decode the encoded function names displayed by the current symbolic debuggers. A full description of the encoding scheme is beyond the scope of this article, however to get a feel for this encoding system, here is a simple example. The following function declaration

```
double someFunc(signed int aInt, float a [illegible]
[illegible]
```

would be encoded as:

```
someFunc__Fifce
```

The encoding is made up of two parts, the function name and the function signature (encoding of arguments). The two parts are separated by "__" (two underscores), and it is recommended that you do not use "__" in your own function names. In our example the first character in the function signature is "F", this indicates that the function is global (or has file scope). This is then followed by four lower case characters, one for each argument, "i" for an int, "f" for float, "c" for char and an "e" for the ellipses. Things get a bit more complicated with user defined types. Comeau C++ comes with a utility called demofile which will read in mangled names from standard input until EOF, and then it will output the mangled names together with the demangled equivalents (see the README file on disk one of the Comeau C++ distribution for more information on demofile). For a more detailed description of function name encoding see the following: ARM page 122-127, The Comeau C++ User's manual pages 16-17.

Now this function name encoding while necessary for C++ functions will cause problems if you are trying to call C functions from your C++ program. To be able to call a C function in C++ you have to tell the compiler that the function is a C function so that the compiler will know not to change the name. You do this by use of a linkage specifier in an extern declaration

```
extern "C" int someFunction(); // declare a C function
```

The "C" tells the compiler that someFunction is a C not a C++ function. Now to do this for all AmigaDOS functions would be a bit laborious to say the least! There is an easier way; an extern statement can be applied to an include file to indicate that it contains C function definitions:

```
extern "C" { [illegible]
[illegible] #include <intuition/intuition.h> }
```

You can even avoid this with Comeau C++ as it comes with a utility, c10/include, which will make C++ versions of all C include files. Basically it creates a C++ header file of the same name as the C version but which contains an extern "C" statement which then includes the C version of the header file. For example the C++ version of exec/types.h on my system contains:

```
extern "C" {    #include "gpp:cc/2.0/include/exec/types.h"
}
```

If you want to create header files to use in C and C++ programs you can make use of the macro constant __cplusplus.

```
#ifdef __cplusplus        extern "C" {          #endif
       /* function declarations */
#ifdef __cplusplus        }         #endif
```

Another common problem encountered when using C++ in a C environment is that C++ requires that a function be declared before it can be used. If C++ encounters a function call prior to a definition or declaration of that function, it will report an error. You must ensure that you include the appropriate header files containing function declarations for all functions that you call. The 2.0 Native Developers kit available from Commodore (Part Number NATDEV20), contains header files which have declarations or prototypes for all Amiga Libraries. These can be found in the include/clib directory. These header files are not C++ ready and you will have to use an extern "C" wrapper when including them in your C++ program.

In C++ as well as overloading your function names you can also have several declarations of your functions which specify default values for some or all of the arguments. These default values are substituted for missing trailing arguments. For instance the following declaration is for a function with two default values:

```
void moveTo( int x = 0, int y = 0);
```

The following are all valid calls to this function.

```
moveTo();     // moves to x = 0, y=0     moveTo(1);    //
moves to x = 1, y=0     moveTo(1,4); // moves to x = 1, y = 4
```

It is only possible to define default values for trailing declarations. The following is not a valid function declaration:

```
void f(int x = 0, int y);
whereas
void f(int x, int y = 0);
```

is legal.

It is also only possible to call a function with trailing arguments missing. The following which you might think would be equivalent to moveTo(0,4) is illegal.

```
moveTo( , 5);   // illegal in C++
```

In general I think that default values should be avoided. I always believe that when programming you should be as specific as possible and the idea that you can leave out arguments to a function just because the compiler will fill them in with default values goes a bit against the grain. However, one use for default values that I would condone would be their use to support backwards compatibility to new versions of library routines. Suppose you had a library function called plopWindow which accepted four arguments, x, y, width and height:

```
boolean plopWindow( int x, int y, int width, int height);
```

which had been around for a long time and is used in a lot of code. You now decide you need to add some extra functionality to the function which requires an extra argument. You have a number of options: you could create a new function called newplopWindow, you could go around all your old code and modify all calls to the function or you could use default arguments in the new version of the function and its declarations:

```
boolean plopWindow(int x, int y, int width, int height,
int doHicky=0);
```

Old code which calls plopWindow with only four arguments when recompiled with the above declaration will pick up a default value of 0 for the fifth argument doHicky. This assumes that a value of 0 for doHicky will produce the old behaviour of plopWindow.

Default value function declarations can be combined with scoping rules to allow the behaviour of function calls to be changed somewhat in different scopes:

```
extern void setPoint(int x, int y);
void sillyFunc( void ) {         setPoint(1,2);  }
extern void setPoint(int x = 0, int y = 0);
setPoint();
```

g++ issues warning messages on the second call to setPoint.

While using default values it is possible to change the way functions are called within different scopes, you cannot have functions defined within functions, as you can in such languages such as Pascal. C++ like C is not really a block based language in the same manner as Pascal. However, both C++ and ANSI C have had, if you like, their blockiness extended at least with regards to variables with the introduction of new scope rules for variables.

In K&R C the scope of variables could be local to a function, local to a file (compilation unit) or to the whole program. It is now possible in both C++ and ANSI C to define variables which are local to a block. A block being those statements contained within curly brackets (a block is also, more formally, called a compound statement).

In ANSI C the declarations of any variables in a block must appear before any non-declaration statements. C++ extends this so that a variable can be declared anywhere within a block prior to its first use (the DICE C compiler also supports this feature even though it is not in the ANSI C standard). This feature was added to allow variables to be declared when an initial value for the variable became available. Blocks can of course be nested, here is a C++ example of some nested blocks with each block having its own declaration of the variable i (DICE will also compile this example):

```
void main(void)    {         printf("Entering block
1\n");
     int i = 100;      //declare first i
     {  // start another block            printf("  Entering
block 2\n");
        int i = 25;  // declare second i
        {  // start another block                  printf("
Entering block 3\n");
          // 3rd i declared in "for" statement
for (int i=0; i < 4; i++)               printf("    i=%d\n", i);
          printf("    Exiting block 3\n");            }
       printf("  i=%d\n", i);               printf("  Exiting
block 2\n");          }  // end block
     printf("i=%d\n", i);          printf("Exiting block 1\n");
     exit(0);     }
```

If this function was called it would produce the following as output:

```
Entering block 1          Entering block 2          Entering block 3
i=3          Exiting block 3          i=107          Exiting block 2
i          i=105          Exiting block 1
```

Note that changing the value of i in a block does not effect the value of i in any other block. Just like changing a variable local to a function does not change the value of a global variable with the same name. In the above code also notice that the third i is initialized in the for statement itself.

If you declare variables in blocks as described above, be careful when modifying code. Removing a declaration of a variable within a block, but forgetting to remove a reference to that variable may not necessarily generate a compiler error if the variable name is also defined in an outer block. Strange errors can result.

A potential problem which could occur in C programs compiled with a C++ compiler results from C++'s automatic creation of a type for a tagged struct. In C++ it is possible for a structure defined in an inner scope to conceal a variable of the same name as the structure in an outer scope. Consider the following (which is based on the example on page 401 of the ARM)

```
#include <stdio.h>
int x[99];
int main(int argc, char* argv[])     {     struct x {int a;};
printf("the size is %d\n", sizeof(x));     }
```

If the above is compiled by a C++ compiler the resulting program will print out "the size is 4". If it were compiled with an ANSI C compiler it will output "the size is 396". In the ANSI C case the sizeof refers to the array. If you wanted to reference the struct the sizeof would have to be changed to sizeof(struct x). In C++, however, because a new type, x, is created when the struct is defined it hides the array. If the sizeof was meant to reference the array then it could be rewritten as sizeof(::x). What may you ask does the :: do?

In K&R and ANSI C, if a function had a local variable with the same name as a global variable the function would not be able to access the global variable directly. Put another way, you usually do not give your local variables the same names as global variables if you need to access the global variables. Well C++ has a remedy for that. A new operator was introduced in C++, ::, which is called the scope resolution operator. This allows a function to access variables of file scope (global) even if that function has a local variable of the same name. For example

```
extern int do_one(void);          extern int do_two(void);
int error = 0; // declare global variable
void do_stuff(void)          {          int error = 0; //
declare local variable
          if ((error = do_one()) != 0)          ::error =
do_two();
          if (error != 0)          ::error = error +
ERR_DO_STUFF_OFFSET;     }
```

The above function will, if it gets an non-zero return value from either of the two functions it calls, set the global variable error to the returned error value plus an offset value. As a matter of style some C++ programmers always use the scope resolution operator when referencing a global variable. This aids people reading the code in letting them know that a global variable is being accessed.

As stated above, the scope resolution operator can only be used to access file scope variables; that is, global variables. In the nested blocks example discussed earlier you could not use this operator from an inner block to access the variable i in an outer block.

The final C++ feature that we will look at in this article is anonymous unions. In C++, unlike K&R and ANSI C, each union does not have to have a name. The fields of such an unnamed or anonymous union must be unique from the names of variables or the names of the fields in other anonymous unions within its scope. An example of a definition of an anonymous union:

```
          union {               int packInteger;
char packBytes[sizeof(int)];          };
```

packInteger and packBytes can be referenced just like ordinary non-member variables eg:

```
          packBytes[0] = 2;
```

In K&R and ANSI C you would have have had to use

```
          union {               int Integer;               char
Bytes[sizeof(int)];          } Pack;
```

and accessed the union members via Pack eg:

```
          Pack.Bytes[0] = 2;
```

If you define a global anonymous union it must be declared as static; that is, a global anonymous union cannot be accessed outside of its compilation units. Failure to declare a global anonymous union as static will result in a compile time error. g++ does complain if a global anonymous union is not declared static, however it crashes my machine while compiling a trivial example containing a global static anonymous union.

Due to the addition of new features to C++ over ANSI and K&R C a number of new reserved keywords were introduced. These keywords cannot be used as variable names, type names etc. Here is a list of the new keywords:

```
asm       catch     class     delete    friend   inline     new     operator
private   protected   public    template       this     throw     try       virtual
```

These may cause problems when including C header files into your C++ programs as it is possible that these keywords could have been used as C argument names, type names or function names. If you do come across such instances then they will have to be changed.

The stronger type checking present in C++ is at times a two edged sword. On the one hand it brings your attention to possible and real errors (and the potential for error). While on the other hand it can sometimes seem to be a nuisance, having the compiler always complaining about assignments etc which seem perfectly valid. I personally prefer a stronger type checked language. Anything that can help me eliminate errors early is well worth the inconvenience of some extra typing. As well as stronger type checking, C++ is also stricter about the use of user defined and built in types than C. This has resulted in a couple of differences in implementation worth watching out for:

1) Character constants have a size of char in C++ — (sizeof('a') will equal 1) the size of int in C (sizeof('a') will equal sizeof(int)). Note in general I have not been caught by problems with C and C++ treatment of char, except in the above case.          2) Enumerations are treated differently in

and C++. In our example of enum days, in C sizeof(Monday) is equal to sizeof(int), in C++ it is equal to sizeof(days) which may or may not equal sizeof(int).

That sums up all the C++ features that I wish to cover in this article. I hope you agree that they supply some useful and well needed extensions to C. Now there is a lot more to C++ than I have covered in this article. I have only discussed the features which could be thought of as simple enhancements to C. In Wiener and Pinson's book they refer to these features as "How C++ Enhances C in Small Ways". Well there are any number of larger ways in which C++ enhances C but they are left for you to discover or for possible future articles.

I have included a bibliography which consists of the various books I referenced during the writing of this article, which cover C and C++. If you are interested in adding a couple of books on C++ to your library I highly recommend the ARM and Coplien's book "Advanced C++". Also, while this article did not cover Object Oriented Programming anybody who seriously wants to use C++ will require knowledge of OOP concepts. To this end I have included a reference to Timothy Budd's excellent book "An Introduction to Object-Oriented Programming".

## ANNOTATED BIBLIOGRAPHY

### K&R C

"The C Programming Language" by Brian W. Kernighan and Dennis M. Ritchie, first edition, 1978, Prentice-Hall. ISBN 0-13-110163-3 The classic book on C programming. Deserves to be one of the best selling computer books around. Divided into two sections; a tutorial section and a language reference section.

### ANSI C

"The C Programming Language" by Brian W. Kernighan and Dennis M. Ritchie, second edition, 1988, Prentice Hall. ISBN 0-13-110362-8 This is a rewrite of their original book to cover the new ANSI version of C.

"The Waite Group's Essential Guide to ANSI C" by Naba Barkakati, 1988, Howard W. Sams & Company. ISBN 0-672-22673-1 A great little book! This is a quick reference guide to ANSI C and the standard library. Well organized and written.

"Standard C" by P.J. Plauger and Jim Brodie, 1989, Microsoft Press. ISBN 1-55615-158-6 Another quick reference guide. I don't like this as much as the "Essential Guide", mainly for style reasons. However, it does have detailed descriptions of each of the header files in the standard library and a useful table indicating what header file to include to pick up definitions of specific types and functions.

### C++

"The Annotated C++ Reference Manual" by Margaret A. Ellis and Bjarne Stroustrup, May 1992, Addison-Wesley. ISBN 0-201-51459-1. I have to emphasize that this book is a reference manual, not a tutorial. However, as reference books go this one ranks among the best. The supplied "commentary" is a good read in itself and explains why certain features are implemented the way they are and, more usefully, what pitfalls to watch out for. This is not the sort of book you are likely to read from start to finish but if you intend to do lots of C++ programming it will prove invaluable. This book is the ANSI base document for the ANSI standardization of the C++ language. In various C++ literature this book is referred to as the ARM.

"Advanced C++: Programming Styles and Idioms" by James Coplien, 1992, Addison-Wesley. ISBN 0-201-54855-0 Once you have the basics of C++ under your belt this is the book to get. People with little formal background in computer programming may find the discussion in some parts a bit of a tough slog but this is a book where the more effort expended the greater the rewards. It includes an appendix on interfacing C and C++ code which to some extent inspired this article. For people interested in moving to object oriented programming the chapters on Object-Oriented Programming and Object Oriented Design are well worth reading. Anybody really serious about programming in C++ should own this book.

"An Introduction to Object Oriented Programming and C++" by Richard Wiener and Lewis Pinson, 1988, Addison-Wesley. ISBN 0-201-15413-7 This book suffers, in my opinion, from a poor layout and choice of typefaces. It also is based on an older version of C++. However, it is written in fairly plain language and, lots of examples and diagrams are used to explain some concepts. Also, unlike some other C++ books, this book does explain some of the terminology and goals of Object Oriented programming before starting into the language itself.

"C++, A Guide For C Programmers" by Sharam Hekmatpour, 1990, Prentice Hall. ISBN 0-13-109471 This book has many examples, the last four chapters are in fact case studies. The solutions to exercises are included. The examples were written with g++. It assumes, as the title suggests, that you have knowledge of C. One thing I didn't like was that it just launched right into the C++ language, no attempt was made to try to explain the goals of object oriented programming and no real outline of C++ and its features was given. Again, this book is based on an older version of the C++ language (and the examples are based on an older g++ compiler) and some of the newer features (such as Templates) are not mentioned. However, there is a new version of the book which comes with disk (PC) containing the source to all the examples. I do not know if this version is based on a newer C++ language definition.

### OOP

"An Introduction to Object-Oriented Programming" by Timothy Budd, 1991, Addison Wesley. ISBN 0-201-54709-0 An excellent book on OOP. Starts off with general concepts and then goes on to show how these concepts are supported in four languages: C++, Objective C, Object Pascal and Smalltalk. Describes the advantages and disadvantages of each language. Good clear examples and case studies. Unfortunately the book was written before templates were introduced to C++, so only a cursory mention of them is made. Well worth getting if you want to understand what OOP is all about.

✔

*Please write to:*
*Paul Gittings*
*c/o AC's TECH*
*P.O. Box 2140*
*Fall River, MA 02722*

# AC'S GUIDE WINTER 1994

Looking for a specific product for your Amiga but you don't know who makes it? Want a complete listing of all the Fred Fish software available? Just looking for a handy reference guide that's packed with all the best Amiga software and hardware on the market today?

If so, you need *AC's GUIDE for the Commodore Amiga*. Each *GUIDE* is filled with the latest up-to-date information on products and services available for the Amiga. It lists public domain software, user's groups, vendors, and dealers. You won't find anything like it on the planet and you can get it only from the people who bring you the best monthly resource for the Amiga, *Amazing Computing*.

So to get all this wonderful information, call **1-800-345-3360** today and talk to a friendly Customer Service Representitive about getting your *GUIDE*. Or stop by your local dealer and demand your copy of the latest *AC's GUIDE for the Commodore Amiga*.

## List of Advertisers


| Company | Page Number |
|---|---|
| Amiga Library Services | CII |
| AMOS | CIV |
| Delphi Noetic Systems | 8 |
| Digital Imagery | 44 |
| SAS | 17 |



# WHAT'S ON IT?

**AC's TECH 4.2 Disk Includes Source & Executables For:**

- True F-BASIC
- Huge Numbers Part 2
- Better Way to C
- AmigaDOS Shared Libraries
- Compression
- A Date with True BASIC
- Programming the Amiga in Assembly

**AND MORE!**

AC's TECH 4.2 CHECK IT OUT!


DJcomputers.cz

# AC's TECH Disk

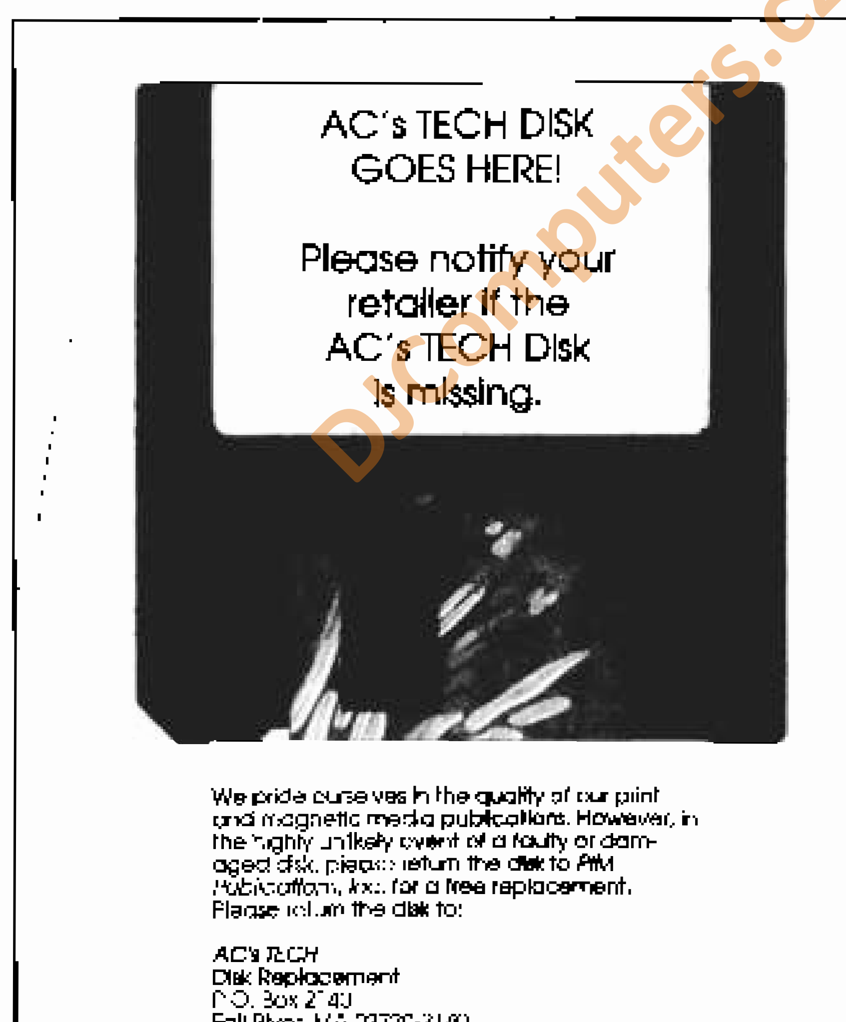
## Volume 4, Number 2

### *A few notes before you dive into the disk!*

- You need a working knowledge of the AmigaDOS CLI as most of the files on the AC's TECH disk are only accessible from the CLI.

- In order to fit as much information as possible on the AC's TECH Disk, we archived many of the files, using the freely redistributable archive utility 'lharc' which is provided in the C: directory. These archive files have the filename extension .lzh.

  To unarchive a file *foo.lzh*, type *lharc x foo*
  For help with lharc, type *lharc ?*
  *Also, files with 'tool' icons can be unarchived from the WorkBench by double clicking the icon, and supplying a path.*




*Be Sure to Make a Backup!*

## CAUTION!

Due to the technical and experimental nature of some of the programs on the AC's TECH Disk, we advise the reader to use caution, especially when using experimental programs that initiate low-level disk access. The entire liability of the quality and performance of the software on the AC's TECH Disk is assumed by the purchaser. PiM Publications, Inc., their distributors, or their retailers, will not be liable for any direct, indirect, or consequential damages resulting from the use or misuse of the software on the AC's TECH Disk. (This agreement may not apply in all geographical areas.)

Although many of the individual files and directories on the AC's TECH Disk are freely redistributable, the AC's TECH Disk and the collection of individual files and directories on the AC's TECH Disk are copyright PiM Publications, Inc., and may not be duplicated in any way. The purchaser, however, is encouraged to make an archival backup copy of the AC's TECH Disk.

Also, be extremely careful when working with hardware projects. Check your work twice, and avoid any damage that can happen. Also, be aware that doing these projects may void the warranties of your computer equipment. PiM Publications, Inc. is not responsible for any damages incurred while attempting this project.

DJcomputers.cz

# HUGE NUMBERS

**By Michael Greibling**

## Introduction

Last article I covered the basic arithmetic operations of addition, subtraction, multiplication, and division and described a practical application of ExNumbers in a CLI-based calculator.

This article extends the ExNumber library functions by introducing an ExInteger module which performs logical, bit, and shift operations (e.g., AND, BSET, SHR, etc.) on ExNumbers and can also convert an ExNumber to/from various based representations (e.g., hexadecimal, binary, octal strings). I also add an ExMathLib0 module which provides an ExNumber interface to the Amiga's built-in IEEE 64-bit math library. Finally, all these new functions are integrated into the CLI-based calculator from last time to vastly improve its functionality.

## ExIntegers

The most obvious extension for the ExNumbers is a way of performing logical operations on them. Doing so requires that we restrict the vast dynamic range (i.e., -9.9E10000 to 9.9E10000) of the ExNumbers into a more manageable range that can be exactly represented within the 52 digits or so of ExNumbers. As well, to give easily discernable ExInteger limits when performing based arithmetic, a further constraint is imposed so that ExIntegers are within the range -(2**172) to 2**172 or -5.98E51 to 5.98E51. ExIntegers can thus exactly represent any 172-bit integer.

The simplest implementation of ExIntegers to support logical operations in Modula-2 is a mathematical set implementation. (Table 1 shows the relationship between set operations and logical operations.) ExIntegers are built up using an array of the 16-bit set data type (BITSET). Figure 1 illustrates the structure of the ExInteger data type.

## ExInteger Conversions

To give us logical operations on ExNumbers several conversion routines are defined which translate ExNumbers into ExIntegers and vice versa. These conversions are hidden from the user of the ExInteger functions (Listing 1) so that the parameters which are passed in and out of the procedures are always seen as ExNumbers. Thus, the ExInteger package interface is simplified so users don't have to perform explicit conversions every time they wish to perform a logical operation on ExNumbers. We see later that this calling convention simplifies the interface to the Calculator as well.

The ExNumbToExInt procedure near the end of Listing 1, converts ExNumbers to ExIntegers. This algorithm first constrains the ExNumber to the valid ExInteger range (i.e., -(2**172) to 2**172). Next, a loop generates ExInteger set elements by taking the modulo 2**16 remainder of the ExNumber to effectively strip out a 16-bit chunk of the ExNumber and then type-casts this number into a 16-bit set (LONGBITSET), stored in the ExInteger. After each loop iteration, the ExNumber is divided by 2**16 and truncated to an integer to give access to the next 16-bit chunk. This loop terminates when all the ExNumber Quads are zero.

The reverse operation of converting ExIntegers to ExNumbers is performed by the ExIntToExNumb procedure. A similar loop scans through the ExInteger chunks, in reverse order (i.e., from highest to lowest), converts each set into a 16-bit unsigned number, multiplies the accumulated total by 2**16, and adds the converted number to the total. The conversion is complete as soon as each ExInteger set has been addressed.

## Logical Operations

The ExInteger module performs the standard logical operations of AND, OR, XOR, NOT or one's complement, bit setting, bit clearing, bit toggling, logical shifts, arithmetic shifts, and rotations. These functions are grouped according to the algorithm which implements each operation. For example, the AND, OR, XOR and NOT functions all call the LOp procedure to perform the detailed logical processing of the ExInteger. For this reason, I just describe the central procedure for each grouping (i.e., LOp in this case) with the understanding that the other procedures which also use this algorithm have similar properties.

The ExAnd procedure serves as the representative of the first grouping which calls the LOp procedure, by passing in a customization function (the And function) which returns the intersection (equivalent to logical AND) of two BITSET arguments. The LOp procedure first translates the two operands to ExIntegers; then the passed function is used, on a 16-bit chunk basis, to logically AND (in this case) together both ExInteger arguments. The result is converted back to an ExNumber and is returned to the ExAnd procedure.

The second class of operations uses the LBit procedure to perform single-bit manipulations such as setting, clearing and toggling bits. The ExSetBit procedure is used as an example to illustrate the general bit algorithm. As before, the ExNumber is converted to an ExInteger. LBit then makes use of the power procedure 'xtoi' from the ExMathLib0 module (described below) which implements the raising of the ExNumber, x, to the ith integral power, where i is an integer. The xtoi routine is used here to produce a single-bit mask based on the principle that $2^n$ sets the nth bit of an integer. In this case, the bit mask is ORed with the ExInteger, using the passed 'Oper' function in a call to LOp. Consequently, the ExNumber returned by this procedure, after conversion from an ExInteger, has its nth bit set.

Shifting operations are more awkward in an ExInteger format so they are implemented as multiplications and divisions by powers of two on ExNumbers. There are three different flavours of shifting algorithms: signed or arithmetic shifts, unsigned or logical shifts, and rotations. Figure 2 shows how these shifting operations differ.

The simplest shifting operation is the logical shift as implemented by the LShift algorithm (Listing 1). The ExNumber is first constrained to a valid ExInteger range. Next, if the bit shift quantity is greater than MaxBase2Bits (172), a zero is returned and the algorithm is aborted since the number has been shifted out of the ExInteger number range; otherwise, a shift mask is calculated using xtoi and, for the ExShl procedure, is multiplied times the number to be shifted. This shifting operation is characterized by the equation $Result = n * 2^b$, where n represents the number to be shifted and b represents the number of bit positions to be shifted.

The rotation operations, implemented by the LRotate procedure, are slightly more complicated because the bit which is rotated out of the ExInteger range must be wrapped around and shifted back into the ExNumber. To help sense the state of a given bit in an ExNumber, the IsBitSet function was created. It forms a mask for a selected bit, ANDs this mask with a number, and then returns true if the bit was set. For ExROR, LRotate calls this function to extract the least significant bit before shifting the number. After the shift, if the detected bit was set, the most significant bit of the result is set using the ExSetBit procedure. The above process is repeated until as many bits have been rotated as were specified by the 'bits' parameter. Note: The worst case shift has been reduced, using a modulo operation, to the number of bits in an ExInteger since rotations always preserve the original number.

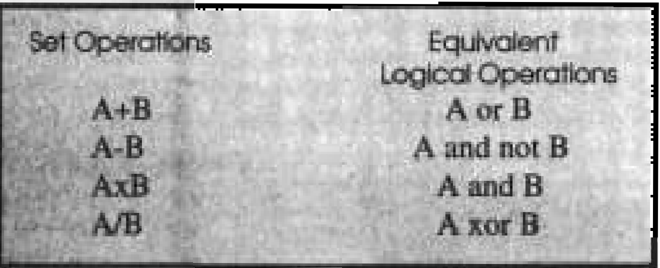
The final shifting procedure, ExAshr, performs an arithmetic shift right on an ExInteger. What this means is that the sign bit of the ExInteger is replicated each time the ExInteger is shifted right so that the number's sign is preserved. Since ExNumbers are implemented with a separate sign bit, this value is easily extracted by setting a SavedBit flag if the sign is negative. The ExInteger is then shifted right one bit at a time (using ExDiv by two) until 'bits' have been shifted. For each shift, if the SavedBit flag was set, the upper bit of the ExInteger is set using ExSetBit to restore the number's sign.

## Based String Conversions

To enable the calculator to deal with numbers in other bases (e.g., hexadecimal, binary, and octal) I need to introduce two new procedures called StrToExInt and ExIntToStr. The first of these routines converts a based string into an ExNumber and the second routine performs the inverse operation of converting an ExNumber into a based string. Both procedures work only with integers: StrToExInt returns an illegal number error if requested to convert a floating point string and ExIntToStr constrains the ExNumber to a legal integer range before performing a conversion. I won't go into the details of these algorithms since they are very similar to the earlier string conversion routines you saw in the ExNumbers module. The chief difference is that the divisor becomes a power of the conversion base instead of a power of ten. Listing 1 has the complete source code for StrToExInt and ExIntToStr if you are interested in the algorithms used for these conversions.

## An ExNumber Math Library

Everyone knows that a calculator has transcendental (e.g., sin, cos, tan), logarithmic (e.g., log, ln), and power (e.g., $x^y$) functions. But these operations are usually very costly in terms of performance and ➜



| Set Operations | Equivalent Logical Operations |
|---|---|
| A+B | A or B |
| A-B | A and not B |
| AxB | A and B |
| A/B | A xor B |

Table 1: Set Operations vs Logical Operations

|  | Chunk Index | 16-Bit Set Elements (Chunks) |
|---|---|---|
| Highest | 0 | 0100 1001 1001 0110 |
|  | 1 | 0000 0010 1101 0010 |
|  | 2 | 0000 0000 0000 0000 |
|  | • | • |
|  | • | • |
|  | • | • |
| Lowest | 10 | 0000 0000 0000 0000 |

Figure 1: ExInteger representation of the number '1234567890'.

DJcomputers.cz

have algorithms which can become very complicated—especially since our calculator has up to 52 digits of precision. In fact, during my literature search, the best algorithms I could find had only from 16 to 23 digits of accuracy. There were a number of alternatives: 1) come up with the algorithms from scratch which would give 52 digits of accuracy; 2) use a lower-accuracy algorithm; 3) use existing lower-accuracy functions from the Amiga's IEEE math libraries. I opted for the third choice since I didn't have the time to invest in producing and testing the required precision algorithms and the speed penalty would be horrendous. As well, there was no point in reinventing the wheel when algorithms of comparable precision already existed on the Amiga.

I essentially created an interface (ExMathLib module in Listing 2) to the double-precision IEEE floating point library. The calculator could thus have 16 digits of precision at hardware speeds (if you have a floating point coprocessor). Several functions such as square root, cube root, integral powers/roots, and factorial do have the full ExNumber precision because the algorithms were easily extended to give 52-digit accuracy. If you have the ability and time to extend the precision of any other functions, I would appreciate hearing from you so that I can update this module with the new algorithms. Any algorithms I receive will be placed in the public domain—with the author's permission.

The heart of the IEEE floating point interface lies in the the ExNumToLongReal and LongRealToExNum conversion routines. To simplify the conversion process and demonstrate the power of reuse, I used several compiler-supplied conversion routines: ConvStrToLongReal which translates a string into a double-precision IEEE floating point number; and ConvLongRealToStr which performs the inverse operation. As well, ExNumToStr and StrToExNum, from the ExNumber module, provide string to ExNumber translations.

An IEEE number is converted to an ExNumber through an intermediate step of translating the number into a string. StrToExNum takes this string and produces a valid ExNumber. To reverse this process and produce an IEEE representation from an ExNumber, ExNumToStr uses an ExNumber to produce a string which ConvStrToLongReal then translates into the double-precision IEEE floating point number. The expX procedure shown below demonstrates the conversion process and the IEEE interface. The expD function is a compiler-supplied library function which ties directly into the Amiga's double-precision IEEE library.

```
PROCEDURE expX (VAR Result : ExNumType; x : ExNumType);  BEGIN
LongRealToExNum (expD(ExNumToLongReal(x)), Result)  END expX;
```

While many routines can be obtained using the IEEE library, some, like the inverse hyperbolic trigonometric operations, are not available in this library. I had to develop these algorithms from their basic definitions which follow:

```
          ArcSinh(x) = Ln(x + Sqrt(x*x + 1))    ArcCosh(x) = Ln(x +
Sqrt(x*x - 1))   ArcTanh(x) = Ln((1 + x) / (1 - x)) / 2
```

where Ln represents the natural logarithm of a number and Sqrt represents the square root of a number.

Several other functions such as integral roots and powers have algorithms which were easily extended to give full 52-digit precision.

The integral root algorithms are based on Newton's iterative method of finding roots of a function whose basic equation is y(n+1) = y(n) - f(y)/f'(y) where y(n+1) is the (n+1)st iterative solution, y(n) is the nth solution, f(y) is the function whose root is required, and f'(y) is the derivative of f(y). I applied this equation to obtain the general root-finding algorithm shown below:

```
y(n+1) = y(n) + (x / y(n)**(r-1) - y(n)) / r
```

where y(n+1) and y(n) are defined as before, r represents the root power (e.g., r = 2 for a square root), and x is the number whose root we wish to determine. The Root procedure (top of Listing 2) implements this general algorithm and also adds the capability of finding negative roots (e.g., the cube root of -8 is -2). Both the sqrtX and rootX exported procedures use this general-purpose Root routine.

Integral powers are calculated using an algorithm published by Donald Knuth in his work, "The Art Of Computer Programming", the second volume. I have adapted his algorithm to also work for negative powers. The resultant implementation is shown in the xtoi procedure in Listing 2. The beauty of Knuth's approach is that the calculation of any integral power involves only about log(n)/log(2) multiplications where n represents the number's integral power. For example, this algorithm calculates x^64 using only six multiplications! The expX and powerX procedures use the xtoi routine whenever they evaluate integral powers.

The last routine for which I have an extended precision algorithm is the factorialX procedure which computes the factorial of a number. Because ExNumbers have a much larger dynamic range (~1x10**-10000 to 1x10**10000) than most other floating point numbers, factorials can be calculated of numbers as large as 3249! Compare this with the typical calculator which only gives factorials as large as 69! However, calculating this large a factorial also normally requires 3248 multiplications which could take quite a while even on the fastest Amiga. To reduce this time, precalculated factorials of 500!, 1000!, 2000!, and 3000! have been stored in the ExMathLib0 module. Thus, to calculate 3249!, only 249 multiplications are required since the algorithm starts with 3000!. Calling a routine 3000 times recursively can take a lot of stack space so the factorial procedure calculates factorials using an iterative algorithm rather than the recursive algorithm everyone is taught in school, to keep the calculator's stack requirements to the CLI default.

## CLI Calculator Revisited

Listing 3 shows the source code for the new CLI calculator which uses all the new functions discussed above and introduces a few new features such as access to 16 ExNumber storage locations, persistent state between invocations, an argument-passing interface, and a choice of numerical display formats. I'll take you on a brief excursion of the features which make up this new calculator.

## Remembering

The first addition is that the calculator now can store and recall up to 16 ExNumbers using the syntax STM n where x represents a number or expression to be stored and n is the location (0 to 15) where the result should be stored by the StoreMemory routine. To recall the number, type Mn where n represents the ExNumber location to recall via the RecallMemory routine. The ExNumbers are stored in a simple array which can be easily extended to allow number storage which is only limited by available memory.

All these storage locations and other calculator state variables are stored in the RAM: drive (via the StoreState procedure) between calculator invocations so results from previous calculations can be reused during later sessions. The persistent memory also helps get around the problem of having expressions which are longer than the maximum allowable input string of 250 characters. They can simply be split up and calculated in pieces, with intermediate results stored in the calculator's memory.

## Argument Interface

In addition to the interactive mode, it is possible to use the calculator much like the Amiga's Eval program where the expression to be evaluated is passed to the calculator when it is invoked. For example, typing 'Calculator 2^10' produces the result 1024. The command line argument is extracted by the GetCLI procedure and then is processed just as if you had typed the expression interactively. Because the memory is retained between calculator invocations, you could store the results of one calculation in memory and then use those results in a following calculation. Remember that command line arguments are automatically separated by the operating system if spaces are left between words so quotation marks must be placed around expressions. For example, to calculate the sum of the first five factorials, from the CLI, type:

```
Calculator "1! + 2! + 3! + 4! + 5!"
```

The answer 153 is displayed when the CLI prompt returns. Of course, the spaces are optional, and 1!+2!+3!+4!+5! (with no spaces) would produce the same result without requiring quotations.

## Output Formats

You can toggle the format of the calculator's output numbers between the default floating point notation and scientific notation. Just type SCI to toggle between these two modes. Be careful to type the exact name as shown because all the calculator functions are case-sensitive. If you type in a number like 2 and then switch to scientific notation you may be shocked at all the trailing zeros that get displayed. Several commands let you suppress these extra digits: DP n lets you enter the number of decimal point digits which should be displayed where n can be a number between 0 and 52. Specifying a value of 0 selects the default floating decimal point notation while any other number fixes the decimal point at n digits. DIG n selects the number of digits that the calculator uses when performing its calculations. Valid values for n are from 1 to the default of 52. All calculator format definitions are saved to the RAM: disk between calculator sessions.

## Other Functions

The calculator allows evaluation of any trigonometric function (SIN, COS, TAN). The default angular units are in degrees. The command DRG toggles between angular units of degrees, radians, and grads.

Based numbers, as discussed above, represent a subset of the ExNumbers. To get the calculator into the based number mode, type BAS n where n represents the numerical base from 2 to 16. The value for n is always specified in decimal notation no matter which base the calculator is using. Numbers containing decimals and exponents are illegal when in a numeric base other than 10. Based numerical systems greater than 10 use the uppercase alphabetic characters A-F to represent numbers in addition to the standard digits but every number must

always begin with a valid base digit from 0 to 9; numbers beginning with the digits A-F require a leading 0. Underscores, apostrophes, and commas may be used to separate groups of digits no matter which based representation is being used.

Table 2 summarizes the calculator operations and commands along with the required syntax when accessing the calculator from the CLI. An additional restriction is that the longest allowable expression string cannot exceed 250 characters.

### Summary

The addition of ExInteger shifting and bit manipulation operations along with an IEEE double-precision mathematical library interface has transformed the original CLI calculator into a full-featured tool which is as powerful as commercial calculators, lacking only a polished graphical interface. In the final article of this series, I introduce a Cascade application program which provides a slick, graphical interface to the CLI-based calculator. Happy computing until then!

## Listing One

```
IMPLEMENTATION MODULE ExIntegers;

(* Some functions to perform bit manipulation on
 ExIntegers.
   This module deals with integral ExNumbers in the
range
   from -2147483648 to 2147483647. Any numbers outside
this
   range are represented with the maximum (or minimum)
   ExNumber from this range.
*)

FROM Conversions IMPORT ConvertToString, ConvertFromString;
FROM ExMathLib0 IMPORT x10y;
FROM ExNumbers IMPORT ExNumType, ExChgSign, ExAbs,
ExMax,
                    GetBaseDigits, ExMin, ExNumb,
DigitsType,
                    ExAdd, ExSub, ExMult, ExDiv,
ExZero,
                    ExOne, ExInit, ExNormalize,
ExStatusType,
                    ExExpHalf, ExDiv10, ExSetTrunc,
                    ExTrunc, ExAdd, ExShift,
                    ExToLongReal, WriteExNum;
FROM InOut IMPORT WriteString, WriteLn,
WriteLongInt,
                    WriteCard;
FROM Strings IMPORT Insert, Delete, Length;

CONST
  MaxBase2Bits = 32;  (* 2147483647 / 1.0E10 *)
  LogicalSize = MaxBase2Bits DIV 8;
  Left = FALSE;
  Right = TRUE;

TYPE
  LogicalType = ARRAY [0..LogicalSize] OF BITSET;
  LogicalProc = PROCEDURE(BITSET, BITSET) : BITSET;
  ExNumbProc = PROCEDURE(VAR ExNumType, ExNumType,
ExNumType);

VAR
  LogZero   : LogicalType; (* All bits cleared or 0 *)
  MaxNumber : ExNumType;   (* 2 ** MaxBase2Bits - 1 *)
  MinNumber : ExNumType;   (* -2 ** MaxBase2Bits - 1 *)
  Two       : ExNumType;   (* The value = 2 *)
  Cnt       : CARDINAL;


(*                             *)
(* Local bit manipulation functions. *)

PROCEDURE And (op1, op2 : BITSET) : BITSET;
BEGIN
  RETURN op1 * op2;
END And;

PROCEDURE AndNot (op1, op2 : BITSET) : BITSET;
BEGIN
  RETURN op1 - op2;
END AndNot;

PROCEDURE Or (op1, op2 : BITSET) : BITSET;
BEGIN
  RETURN op1 + op2;
END Or;

PROCEDURE Xor (op1, op2 : BITSET) : BITSET;
BEGIN
  RETURN op1 / op2;
END Xor;


(*                             *)
(* Miscellaneous local procedures *)

PROCEDURE Max (x, y : INTEGER) : INTEGER;
BEGIN
  IF x > y THEN
    RETURN x;
  ELSE
    RETURN y;
  END;
END Max;


PROCEDURE ConstrainExNum (VAR Number : ExNumType);
(* Limit Number to be within MinNumber to MaxNumber and
   eliminate any fractional portions. *)
BEGIN
  ExMin(Number, MaxNumber, Number);
  ExMax(Number, MinNumber, Number);
  ExTrunc(Number);
END ConstrainExNum;


PROCEDURE ExNumToLogical (Numb        : ExNumType;
                          VAR Logical : LogicalType);
VAR
  CurState : ExNumType;
  Scale    : ExNumType;
  Temp     : ExNumType;
  Temp2    : ExNumType;
```

```
  LogCnt    : INTEGER;
BEGIN
  (* Constrain op1, op2 to be within the logical number
set *)
  ConstrainExNum(Num1);

  (* translation scaling factor *)
  ExNumb(65536, 0, 0, Scale);
  ExDiv(DivScale, Ex1, Scale);

  (* perform conversion *)
  LogCnt := 0;
  Logical := LogZero;
  WHILE NOT IsZero(Num1) DO
    ExDiv(Temp2, Num1, DivScale);
    ExTrunc(Temp2);
    ExMult(Temp, Temp2, Scale);
    ExSub(Temp, Num1, Temp);
    IF LogCnt > LogicalSize THEN RETURN END;
    Logical[LogCnt] := BITSET(ExToLongInt(Temp));
    Num1 := Temp2;
    INC(LogCnt);
  END;
END ExNumToLogical;

PROCEDURE LogicalToExNum(Logical : LogicalType;
                         VAR Numb : ExNumType);
VAR
  Scale   : ExNumType;
  Temp    : ExNumType;
  LogCnt  : INTEGER;
BEGIN
  (* translation scaling factor *)
  ExNumb(65536, 0, 0, Scale);

  (* perform conversion *)
  Numb := Zero;
  FOR LogCnt := LogicalSize TO 0 BY -1 DO
    ExMult(Numb, Numb, Scale);
    ExNumb(LONGCARD(Logical[LogCnt]), 0, 0, Temp);
    ExAdd(Numb, Numb, Temp);
  END;
END LogicalToExNum;


(*-----------------------*)
(* Local procedure to perform general    *)
(* logical operations on ExNumbers.      *)

PROCEDURE Op(VAR Result : ExNumType;
             op1        : ExNumType;
             Oper       : LogicalProc;
             op2        : ExNumType);
VAR
  i : CARDINAL;
  Log1, Log2 : LogicalType;
BEGIN
  (* Translate to logicals *)
  ExNumToLogical(op1, Log1);
  ExNumToLogical(op2, Log2);

  (* Operate on Log1 and Log2 one word at a time *)
  FOR i := 0 TO LogicalSize DO
    Log2[i] := Oper(Log1[i], Log2[i]);
  END;

  (* translate back the result *)
  LogicalToExNum(Log2, Result);
END Op;


(*-----------------------*)
(* Local procedure to perform general  *)
(* single bit operations on ExNumbers. *)

PROCEDURE OBit (VAR Result : ExNumType;
                number     : ExNumType;
                Oper       : LogicalProc;
                bitnum     : CARDINAL);
VAR
  Temp : ExNumType;
BEGIN
  (* Constrain number to be within the logical number set
*)
  ConstrainExNum(number);

  (* constrain bitnum from 0 to MaxBase2Bits *)
  IF bitnum > MaxBase2Bits THEN
    (* no bits are changed *)
    Result := number;
    RETURN;
  END;

  (* calculate 2^bitnum *)
  xtoi(Temp, Two, LONGINT(bitnum));

  (* set the bitnum bit position *)
  Op(Result, number, Oper, Temp);
END OBit;


(*-----------------------*)
(* Local function to extract a bit from *)
(* an ExNumber.                          *)

PROCEDURE BitSet (number : ExNumType;
                  bitnum : CARDINAL) : BOOLEAN;
VAR
  Temp : ExNumType;
BEGIN
  (* Constrain number to be within the logical number set
*)
  ConstrainExNum(number);

  (* constrain bitnum from 0 to MaxBase2Bits - 1 *)
  IF bitnum >= MaxBase2Bits THEN
    (* assume zero *)
    RETURN FALSE;
  END;

  (* calculate 2^bitnum *)
  xtoi(Temp, Two, LONGINT(bitnum));

  (* extract the requested bit *)
  Op(number, number, AND, Temp);

  (* return bit to boolean *)
  RETURN NOT IsZero(number);
END BitSet;


(*-----------------------*)
(* Local procedure to perform general  *)
(* shifting operations on ExNumbers. *)

PROCEDURE OShift (VAR Result : ExNumType;
```



DJComputers.cz

```
                    number      : ExNumType;
                    ExOper      : ExNumOpType;
                    bits        : CARDINAL);
VAR
  Temp : ExNumType;
BEGIN
  (* Constrain number to be within the logical number set
*)
  ConstrainExNum(number);

  (* constrain bits from 0 to MaxBase2Bits *)
  IF bits > MaxBase2Bits THEN
    (* shifted out of range *)
    Result := Ex0;
     RETURN;
  END;

  (* calculate 2^bits *)
  ExPwr(Temp, Two, LONGREAL(bits));

  (* shift the number *)
  ExOper(Result, number, Temp);

  (* Constrain number to be within the logical number set
*)
  ConstrainExNum(Result);
END Shift;


(*-----------------------*)
(* Local procedure to perform general  *)
(* rotate operations on ExNumbers. *)

PROCEDURE Rotate (VAR Result : ExNumType;
                      number      : ExNumType;
                      ShiftRight  : BOOLEAN;
                      bits        : CARDINAL);
VAR
  ShiftCnt : CARDINAL;
  SavedBit : BOOLEAN;
  Half     : ExNumType;
BEGIN
  (* Constrain number to be within the logical number set
*)
  ConstrainExNum(number);

  (* constrain bits from 0 to MaxBase2Bits *)
  bits := bits MOD (MaxBase2Bits + 1);
  ExNumb(0.5, 0, Half);

  FOR ShiftCnt := 1 TO bits DO
    IF ShiftRight THEN
      (* save the bit to be shifted *)
      SavedBit := bitSet(number, 0);

      (* shift the number right *)
      ExMult(number, number, Half);
      ExTrunc(number);
      IF SavedBit THEN
       ExSetBit(number, number, MaxBase2Bits - 1);
       END;
    ELSE
      (* save the bit to be shifted *)
     SavedBit := bitSet(number, MaxBase2Bits - 1);

      (* shift the number left *)
     ExMult(number, number, Two);
      (* restore the saved bit *)
      IF SavedBit THEN
        ExSetBit(number, number, 0);
        END;
      END;

  END;

  (* Constrain number to be within the logical number set
*)
  Result := number;
  ConstrainExNum(Result);
END Rotate;


(*-----------------------*)
(* Exported procedures.                *)

PROCEDURE ExAnd (VAR Result : ExNumType;
                     op1, op2   : ExNumType);
BEGIN
  LOp(Result, op1, And, op2);
END ExAnd;


PROCEDURE ExOr (VAR Result : ExNumType;
                    op1, op2   : ExNumType);
BEGIN
  LOp(Result, op1, Or, op2);
END ExOr;


PROCEDURE ExXor (VAR Result : ExNumType;
                     op1, op2   : ExNumType);
BEGIN
  LOp(Result, op1, Xor, op2);
END ExXor;


PROCEDURE ExIntDiv (VAR Result : ExNumType;
                        op1, op2   : ExNumType);
BEGIN
  (* Constrain inputs to be integers *)
  ConstrainExNum(op1); ConstrainExNum(op2);
  ExDiv(Result, op1, op2);
  ExTrunc(Result);
END ExIntDiv;


PROCEDURE ExMod (VAR Result : ExNumType;
                     op1, op2   : ExNumType);
BEGIN
  (* Result := op1 - (op1 DIV op2) * op2 *)
  ConstrainExNum(op1); ConstrainExNum(op2);
  ExIntDiv(Result, op1, op2);
  ExMult(Result, Result, op2);
  ExSub(Result, op1, Result);
END ExMod;


PROCEDURE ExSetBit (VAR Result  : ExNumType;
                        number  : ExNumType;
                        bitnum  : CARDINAL);
BEGIN
  LBit(Result, number, On, bitnum);
END ExSetBit;
```

## Huge Numbers Part II

```
PROCEDURE ExClrBit (VAR Result : ExNumType;
                    number     : ExNumType;
                    BitNum     : CARDINAL);
BEGIN
  Bit(Result, number, AndNot, BitNum);
END ClearBit;


PROCEDURE ExToggleBit (VAR Result : ExNumType;
                       number     : ExNumType;
                       BitNum     : CARDINAL);
BEGIN
  LBit(Result, number, Xor, BitNum);
END ExToggleBit;


PROCEDURE ExOnesComp (VAR Result : ExNumType;
                      number     : ExNumType);
BEGIN
  (* Constrain number to be within the logical number set
*)
  ConstrainExNum(number);
  IF number.Sign = positive THEN
    (* Subtract from the maximum number *)
    ExSub(Result, MaxNumber, number);
  ELSE
    (* Subtract from the minimum number *)
    ExSub(Result, MinNumber, number);
  END;

  (* Complement the sign bit *)
  ExChgSign(Result);
END ExOnesComp;


PROCEDURE ExShl (VAR Result : ExNumType;
                 number     : ExNumType;
                 numbits    : CARDINAL);
BEGIN
  ExShift(Result, number, Left, numbits);

  (* Determine the result sign *)
  IF BitSet(Result, MaxBase2Bits - 1) THEN
    Result.Sign := negative;
  ELSE
    Result.Sign := positive;
  END;
END ExShl;


PROCEDURE ExRol (VAR Result : ExNumType;
                 number     : ExNumType;
                 numbits    : CARDINAL);
BEGIN
  ExRotate(Result, number, Left, numbits);
END ExRol;


PROCEDURE ExShr (VAR Result : ExNumType;
                 number     : ExNumType;
                 numbits    : CARDINAL);
BEGIN
  ExShift(Result, number, Right, numbits);
  ExAbs(Result); (* clear the sign *)
END ExShr;


PROCEDURE ExAshr (VAR Result : ExNumType;
                  number     : ExNumType;
                  numbits    : CARDINAL);
VAR
  ShiftCnt : CARDINAL;
  SavedBit : BOOLEAN;
BEGIN
  (* Constrain number to be within the logical number set
*)
  ConstrainExNum(number);

  (* Constrain bits from 0 to MaxBase2Bits *)
  IF numbits > MaxBase2Bits THEN
    (* shifted out of range *)
    Result := Zero;
    RETURN;
  END;

  (* set the SavedBit to the correct sign *)
  SavedBit := number.Sign = negative;

  (* shift the number *)
  FOR ShiftCnt := 1 TO numbits DO
    (* shift the number right *)
   ExDiv(number, number, Two);

    (* restore the saved bit *)
    IF SavedBit THEN
    ExSetBit(number, number, MaxBase2Bits - 1);
    END;
  END;

  (* truncate any fraction *)
  Result := number;
  ExTrunc(Result);
END ExAshr;


PROCEDURE ExRor (VAR Result : ExNumType;
                 number     : ExNumType;
                 numbits    : CARDINAL);
BEGIN
  ExRotate(Result, number, Right, numbits);
END ExRor;


(*$S-*)
PROCEDURE StrToExNum (S     : ARRAY OF CHAR;
                      Base  : BaseType;
                      VAR A : ExNumType);
VAR
  StrCnt, TenCnt : INTEGER;
  Multiplier     : INTEGER;
  Scale, Temp    : ExNumType;

  PROCEDURE Digit() : LONGINT;
  VAR
    Str : ARRAY [0..1] OF CHAR;
    Digits : LONGINT;
  BEGIN
    (* Extract a digit *)
    Str[0] := S[StrCnt]; Str[1] := 0C;
     INC(StrCnt);

    IF NOT Conv.StrToNum(Str, Digits, Base, FALSE) THEN
      ExStatus := IllegalNumber;
       RETURN 0;
    END;
    RETURN Digits;
```

DJComputers.cz

```
  END TrimStr;

BEGIN
  S := '';
  [illegible]
    Len := LengthStr(S);
  [illegible](Base, 0, 0, Scale);

  (* Skip Leading Blanks *)
  WHILE (Index < Len) AND (S[Index] = ' ') DO INC(Index)
  END;

  WHILE (Index < Len) AND [illegible] DO
    [illegible](Temp);
    ExMult(A, A, Scale);
    ExAdd(A, A, Temp);
  END;
END StrToExtended;
```

| | |
|---|---|
| + | Addition |
| - | Subtraction |
| *, x | Multiplication |
| /, ÷ | Division |
| ² | Squared |
| ³ | Cubed |
| ⁻¹ | Reciprocal |
| () | Brackets |
| ^, ** | Power |
| % | x 0.01 |
| ! | Factorial |
| &, AND, · | Logical And |
| \|, OR | Logical Or |
| XOR | Logical Exclusive Or |
| CPL | Logical Complement |
| MOD | Modulo |
| DIV | Integer Division |
| SQRT | Square Root |
| CBRT | Cube Root |
| ROOT | Any Root |
| e | Natural Log Base |
| eˣ | Power of e |
| LN | Natural Logarithm |
| LOG | Base 10 Logarithm |
| (A)SIN | (Arc)Sine |
| (A)COS | (Arc)Cosine |
| (A)TAN | (Arc)Tangent |
| (A)SINH | (Arc)Hyperbolic Sine |
| (A)COSH | (Arc)Hyperbolic Cosine |
| (A)TANH | (Arc)Hyperbolic Tangent |
| SBIT | Set Bit |
| CBIT | Clear Bit |
| TBIT | Toggle Bit |
| SHR | Shift Right |
| SHL | Shift Left |
| ASR | Arithmetic Shift Right |
| ROR | Rotate Right |
| ROL | Rotate Left |
| Mn | Memory Location n |
| STM n | Store to Memory n |
| Pi | Constant Pi |
| SCI | Toggle Scientific/Floating Pt. |
| BAS n | Change to Base n |
| DIG n | Use n Digits |
| DP n | Use n Decimal Places |
| DRG | Toggle Degree/Radian/Grad |

Table 2: Calculator Operations and Commands

```
PROCEDURE ExtendedToStr (A : ExNumType;
                         Base : BaseType;
                         VAR S : ARRAY OF CHAR);
VAR
  Digit : INTEGER;
  InvScale, Scale, Temp, Temp2 : ExNumType;

  PROCEDURE PutDigits (Numb : CARDINAL);
  VAR
    Str : ARRAY [0..80] OF CHAR;
    Ok  : BOOLEAN;
  BEGIN
    Ok := ConvNumToStr(Str, Numb, Base, FALSE, 4, '0');
     InsertSubStr(S, Str, 0);
  END PutDigits;

BEGIN
  (* Constrain number to be within the machine number set
*)
  ConstrainExNum(A);

  S := '';
  Digit := 0;
  ExNumb(Base, 0, 0, Scale);
  xtoi(Scale, Scale, 4);
  ExDiv(InvScale, ExOne, Scale);

  (* Translate number to a string *)
  REPEAT
    (* Temp := A MOD Scale *)
    ExMult(Temp2, A, InvScale);
    ExTrunc(Temp2);
    ExMult(Temp, Temp2, Scale);
    ExSub(Temp, A, Temp);

    (* Translate to a character *)
    PutDigits(ExToLongCard(Temp));

    (* Reduce A by scaling factor *)
    A := Temp2;
  UNTIL IsZero(A);
END ExtendedToStr;


BEGIN
  (* create the number 2 *)
  ExNumb(1, 2, 0, Two);

  (* Initialize the maximum number *)
  xtoi(MaxNumber, Two, MaxExp);
  ExSub(MaxNumber, MaxNumber, ExOne);

  (* Initialize the minimum number *)
  MinNumber := MaxNumber;
  [illegible](MinNumber);

  (* Initialize the zero logical *)
  FOR Cnt := 0 TO LogicalSize DO
    LogZero[Cnt] := 0;
  END;
END ExIntegers.
```

## Listing Two

```
TYPEREF [illegible] BOOLEAN [illegible];

FROM FxNumbers IMPORT FxNumType, [illegible] FxSub, FxMult,
FxDiv,
                        FxChgSign, FxAbs, FxCompare, [illegible],
                        FxCompareType, [illegible], FxZero,
[illegible],
                        [illegible], [illegible], FxPlus,
[illegible],
                        [illegible], [illegible],
[illegible],
                        FxToLongInt, StrToFxNum, [illegible]
                        [illegible], FxEquals,
[illegible],
                        IsZero, SignType, [illegible];
FROM [illegible] IMPORT [illegible], [illegible], [illegible];
FROM LongRealConversions IMPORT ConvertLongRealToStr,
ConvStrToLongReal;
FROM LongMathLib IMPORT [illegible], [illegible], sin, cos,
[illegible],
                        [illegible], [illegible], [illegible], exp,
ln,
                        [illegible], [illegible], [illegible]
FROM RealTypes [illegible] OpenLongReal, OpenLongRealParams;


VAR
  ToRadians : FxNumType;
  ToDegrees : FxNumType;
  [illegible]500 : FxNumType;
  [illegible]1000 : FxNumType;
  [illegible]2000 : FxNumType;
  [illegible]3000 : FxNumType;


PROCEDURE FxToLongReal (x : FxNumType) : LONGREAL;
VAR
  Num : LONGREAL;
  Str : ARRAY [0..80] OF CHAR;
BEGIN
  (* Convert FxNum into LONGREAL via a string *)
  FxToStr(Str, x, [illegible]);
  IF ConvStrToLongReal(Str, Num) THEN
    RETURN Num;
  ELSE
    RETURN 0.0;
  END;
END FxToLongReal;


PROCEDURE LongRealToFx (x : LONGREAL; VAR Result :
FxNumType);
VAR
  Str : ARRAY [0..80] OF CHAR;
BEGIN
  (* Convert LONGREAL into an FxNum via a string *)
  IF ConvertLongRealToStr(Str, x, [illegible]) THEN
    StrToFxNum(Str, Result);
  ELSE
    Result := [illegible];
  END;
END LongRealToFxNum;


PROCEDURE [illegible] (VAR Result : FxNumType; x : FxNumType; i :
CARDINAL);
```

```
(* [illegible], see Knuth, page 442, The Art Of
Computer Programming, Vol 2 *)
VAR
  Y : FxNumType;
  negative : BOOLEAN;
BEGIN
  Y := [illegible];
  negative := i < 0;
  i := ABS(i);
  LOOP
    IF ODD(i) THEN FxMult(Y, Y, x) END;
    i := i DIV 2;
    IF i = 0 THEN EXIT END;
    FxMult(x, x, x);
  END;
  IF negative THEN
    FxDiv(Result, [illegible], Y);
  ELSE
    Result := Y;
  END;
END [illegible];


PROCEDURE Root (VAR Result : FxNumType;
                    x      : FxNumType;
                    i      : CARDINAL);
(* Use iterative solution of a general root equation *)
VAR
  y, yp, [illegible], [illegible] : FxNumType;
  root : LONGREAL;
  negate : BOOLEAN;
BEGIN
  IF (x.Sign = negative) & (ODD(i) OR (i < 2)) THEN
    Status := [illegible];
    Result := [illegible];
  ELSIF IsZero(x) THEN
    Result := x;
  ELSE
    (* handle negative roots *)
    IF x.Sign = negative THEN FxAbs(x); negate := TRUE
    ELSE negate := FALSE
    END;

    (* estimate of the ith root *)
    root := 1.0D / FLOAT(i);
    LongRealToFx(power(FxToLongReal(x), root),
yp);
    FxCard(i, [illegible], [illegible]);    (* i *)
    [illegible] ([illegible], [illegible]);  (* [illegible] *)

    (* calculate the root *)
    LOOP
      (* y := 1/i * (yp * (i-1) + x / yp^(i-1)) *)
      FxMult(Y, yp, [illegible]);
      [illegible] (y, [illegible]);
      [illegible] ([illegible]);
      FxAdd(y, y, [illegible]);
      FxMult(y, y, [illegible]);
      IF FxCompare(y, yp) = FxEqual THEN EXIT END;
      yp := y;
    END;

    (* adjust the number's sign *)
    Result := y;
    IF negate THEN FxChgSign(Result) END;
  END;
END Root;


PROCEDURE [illegible] (VAR Result : FxNumType; x : LONGINT);
BEGIN
  FxCard([illegible], Result);
```

# Huge Numbers Part II

```
[illegible]

PROCEDURE [illegible]Deg[illegible](VAR [illegible]Angle : ExtNumType);
(* Convert a [illegible] *)
BEGIN
 [illegible]
END [illegible];

PROCEDURE [illegible]ToRad(VAR [illegible]Angle : ExtNumType);
(* Convert a degree [illegible] *)
BEGIN
 ExMult([illegible]Angle, ToRad[illegible], [illegible]);
END [illegible];

PROCEDURE [illegible](VAR Result : [illegible]; x : [illegible]);
BEGIN
 Root(Result, x, 2);
END [illegible];

PROCEDURE [illegible](VAR Result : ExNumType; x : ExNumType);
BEGIN
 [illegible]
END [illegible];

PROCEDURE [illegible](VAR Result : ExNumType; x : ExNumType);
BEGIN
 [illegible]
END [illegible];

PROCEDURE Factorial(VAR prev, current : LONGINT;
                    VAR PrevFact, Result : ExNumType);
(* [illegible] *)
VAR
  i : [illegible];
BEGIN
  FOR i := prev+1 TO current DO
    (* [illegible] *)
    [illegible]
    ExMult(PrevFact, PrevFact, Result);
  END;
  prev := current;
  [illegible] := PrevFact;
END Factorial;

PROCEDURE Factorials(VAR Result : ExNumType; n :
LONGINT);
CONST
  MaxFactorial = [illegible];
VAR
  [illegible] : LONGINT;
  prev : ExNumType;
BEGIN
  IF [illegible] THEN
    ExStatus := [illegible];
    [illegible]
    RETURN;
  END;
  [illegible]
  ELSIF n < 1000 THEN prev := [illegible]
  [illegible]
  [illegible]
  ELSE [illegible]
  END;
  Factorial([illegible], prev, Result);

END [illegible];

PROCEDURE [illegible](VAR [illegible] : ExNumType);
VAR
  [illegible]
BEGIN
  [illegible]
  [illegible]
  IF [illegible] THEN
    [illegible]
  ELSE
    [illegible]
  END;
END [illegible];

PROCEDURE [illegible](VAR [illegible] : ExNumType);
VAR
  [illegible]
BEGIN
  [illegible]
  [illegible]
  IF [illegible] THEN
    [illegible]
  END;
  [illegible]
  END;
END [illegible];

PROCEDURE [illegible](VAR Result : ExNumType; x, y :
ExNumType);
VAR
  [illegible] : LONGREAL;
BEGIN
  [illegible]
  [illegible]
  IF [illegible] THEN
    [illegible]
  ELSE
    [illegible]
  END;
END [illegible];

PROCEDURE [illegible](VAR Result : ExNumType; x : ExNumType);
BEGIN
  [illegible]
END [illegible];

PROCEDURE [illegible](VAR Result : ExNumType; x : ExNumType);
BEGIN
  [illegible]
END [illegible];

PROCEDURE [illegible](VAR Result : ExNumType; x : ExNumType);
BEGIN
  [illegible]
END [illegible];

PROCEDURE [illegible](VAR [illegible] : ExNumType);
BEGIN
  [illegible]
END [illegible];
```

# Huge Numbers Part II

```
[illegible]
```

✔

**Complete source code & listings can be found on the AC's TECH disk.**

**Please write to:**
**Michael Griebling**
**c/o AC's TECH**
**P.O. Box 2140**
**Fall River, MA 02722**

DJComputers.cz

# PROGRAMMING THE AMIGA IN ASSEMBLY LANGUAGE

BY WILLIAM P. NEE

There's a lot to cover in this article. We'll discuss using the math co-processor, a new assembler, another way to tighten up your code and two programs that demonstrate the beauty of Chaos. I'll also show you how to color cycle each of the ten available palettes we'll use. To run this program you'll need at least an Amiga 3000 or a modified Amiga that includes the 68881/68882 math chip.

## 68881/68882

First the co-processor. There are eight additional registers (FP0 to FP7) in the 68882 all designed for high precision, high speed math operations. A special advantage of these registers is that you don't have to open any library to use them so there are no offsets to worry about. Be sure though, to have the MATHIEEEDOUBBAS and MATHIEEEDOUBTRANS libraries available in your Libs: directory. While these two libraries are not called directly by the math co-processor, they are used indirectly by the various commands.

The commands for these registers are very easy to use and mainly involve putting a F in front of most of the single-precision commands I've previously written about (see AC's TECH Volume 2 Number 2). The extensions, however, are a little different since we can now handle numbers in several formats at one time. In addition to the usual .B, .W, and .L extensions we'll use:

| | | | |
|---|---|---|---|
| .S | (single-precision) | .D | (double-precision) |
| .X | (extended precision) | .P | (packed decimal) |

Any operation can be performed on any fp register. While this doesn't sound like such a big deal, remember that with double-precision all commands used d0/d1 or d0/d1 and d2/d3. It not only took four registers to add two numbers, but you could only use the first four; multiple operations required you to save the result in two more registers or use a variable. With the fp registers you can move a value to any register, add any two registers, and add a variable/constant and a register. And they're fast! They're a lot faster than single-precision and, of course, a lot more precise.

## Using the Math Co-Processor

Let's look at some of the more common fp commands.

| | |
|---|---|
| FMOVE.D #1.0123456789,FP2 | - move a double-precision value directly into a fp register |
| FMOVE.D 2(A5),FP1 | - move the double-precision value at 2(a5) into a fp register |
| FMOVE.D FP4,0(A5) | - move the value in the fp register to a double-precision value at 0(a5) |
| FMOVE.X FP0,FP1 | - move the fp value from one fp register to another |
| FMOVE.L FP7,ACROSS | move the integer value in the fp register to a long-word variable |
| FMOVE.L FP6,D1 | move the integer value in the fp register to a data register |

The rest of the commands are variations of the SP or DP commands

| | |
|---|---|
| FADD.D 2(A5),FP1 | add a dp variable and an fp register |
| FSUB.X FP1,FP2 | - subtract two fp registers |
| FMUL.D 4(A5),FP3 | multiply a dp variable and an fp register |
| FDIV.X FP5,FP0 | - divide two fp registers |
| FSIN.X FP1 | - get the sin of the value in an fp register |

Well, you can see how the commands would go. I've included a list of the major commands in Table I although this article doesn't use most of them. If the command uses multiple registers, the second fp register contains the results. So in FADD.X FP0,FP1 fp1 would contain the result of adding the two fp registers; fp0 would still contain it's original value.

## ANOTHER CODE TECHNIQUE

You'll notice that several of the examples above use an offset from register a5 as the variable to add to the fp register. That's because most of the variables in the program for this article are referenced relative to a5. Initially, each variable is equated to it's distance from a starting variable V1 or V2. If the first variable is E it's distance is zero so E EQU 0. In the second half of the program there are several long-word double-precision values. At the end of the program they are stored as:

```
V2
E     DC.D   0,0
N1    DC.D   0,0
N2    DC.D   0,0
...
N11   DC.D   0,0
...
R1    DC.D   0,0
...
```

At the beginning of the program I equate all of these variables to their distance from V2:

```
E    EQU   0
N1   EQU   8
N2   EQU   16
 .
A11  EQU   80
 .
B11  EQU   72
 .
```

Now, all I have to do is include the line LEA V2(PC),A5 and all variables can be referenced by name as an offset from a5. For example, FADD.D B11(A5),FP3 will add the variable B11 to the value in fp3. This cuts down on code length since variables are now considered as an offset instead of having to reference their actual locations. Of course, values can be moved to the variables using the same method. FMOVE.D FP0,E(A5) moves the value in fp0 to the dp variable E. Just remember to equate your variables as a distance from the initial variable, store your variables in order and put the first variable location in an address register (usually a4 or a5) that won't be used again. If the register must be used, keep restoring the variable location.

Look at the difference in code for these four lines.

```
[illegible]                      [illegible]
[illegible]                      [illegible]
[illegible]                      [illegible]
[illegible]                      [illegible]
```

You can see that PHXASS tries to optimize the code and succeeds except in the last case. Using LABEL(A5), however, keeps the code length down to 4 bytes. In a future article we'll see how PHXASS can optimize all four examples.

## ANOTHER ASSEMBLER

Now for a new assembler. This article uses PHXASS and PHXLNK from Fish Disk #459, by Frank Wille. This is the only PD/Shareware assembler I know about that uses the 68881/68882 FPCP. It will also accept the various different commands for the 68000/10/20/30/40. There are some specific things you need to do with your code when using PHXASS. First, if you're going to use the math chip, include the command FPU 1 near the beginning (the "1" is actually optional), and if you're going to use any specific 68020/30 commands, also include MACHINE 68020 or MACHINE 68030 at the beginning. This will, of course, keep your program from running on a 68000/10/20.

Some peculiarities of this assembler are:

1) All include files must be in quotes – this took me several weeks to figure out!

2) References can not include a "." such as in NW.RASTPORT; you must use NW_RASTPORT. I rewrote all of my include files to be written this way.

3) A macro can not begin with "@" to show it is different. In my program I'm using a special PSET so I tried calling it @PSET, but no go. So I deleted GFXMACROS and put all graphics macros at the beginning of the program.

4) There is a problem with some macros that use NARG. If there are no arguments passed, NARG may be 1 instead of 0. Just re-write that portion as code instead of as a macro.

5) The assembler will accept MOVE.L #47,D0 instead of #$47; it only stores the 47, however, in D0.

6) You may have to include SECTION name,CODE at the beginning of your program to avoid an "out of memory" error.

The good news is that all of these bugs and others have been corrected by Mr. Wille. I registered with him by sending him $15.00 and got a new PHXASS version (V3.58) in under two weeks. Since I'll only be using, however, the version on the fish disk, this program is written to compensate for the bugs in that version. I urge you to register with Mr. Wille if you're going to do any assembly language programming. PHXASS is fast! Also, for your convenience, I've included PASB - a script file that will assemble your .ASM file, PHXLNK it and delete the .O file. The DOC files for PHXASS and PHXLNK are also included on the magazine disk.

## CHAOS AND BEAUTY

To demonstrate the use of floating-point numbers I combined two different programs that graphically display Chaos and beauty. Chaos is a fairly new theory that demonstrates how nothing can be certain, that there are always subtle changes in everything that, in the long run, these

DJcomputers.cz

changes still produce an overall effect that can be predicted. Most of the Chaos equations show how you can start with random values and eventually settle down to some type of organized display.

The first program comes from the book "Symmetry in Chaos" by Field and Golubitsky. In their equations using complex numbers (those numbers with i, the square root of -1) they make extensive use of the conjugate of a complex number Z. If Z=X+iY then the conjugate of Z, called Zbar, is X-iY. Z squared is X*X-Y*Y+2iXY while Zbar squared is X*X-Y*Y-2iXY; and Z*Zbar is X*X+Y*Y.

These equations produce very complex but symmetrical patterns. The variables used are: L, A, B, O, G, and N where N is the degree of symmetry, at least 3. Once an original Z with X and Y coordinates has been selected the next Z value is found by the rather formidable equation:

```
[illegible]
```

Not to worry though, this equation can be simplified in Basic as:

```
LOOP
  Zzbar=X*X+Y*Y
  Zreal=X: Zimag=Y
  FOR I=1 TO N-2
    ZR=Zreal*X-Zimag*Y
    Zimag=Zimag*X+Zreal*Y
    Zreal=ZR: Zimag=ZI
  NEXT I
  P=L+A*Zzbar+B*(X*Zreal-Y*Zimag)
  newX=P*X+G*Zreal-O*Y
  newY=P*Y-G*Zimag+O*X
  X=newX: Y=newY: GOTO LOOP
```

What gets plotted is (320+X*Xscale,200-Y*Yscale). The scale factors need to be computed for each set of display values. There are 12 data lines of values near the end of the program for you to experiment with. You can change L through G or just N. If you change N, however, it usually must increase or decrease by multiples of 2. We'll go over this equation in more detail when I discuss the listing.

The other equation is a completely different approach to Chaos. It's from an article by James Yorke entitled "Quasiperiodicity versus Chaos"; even in 1984 when this was written, Chaos was an item of interest. In this program 16 constants are initially selected and then three random constants (E, W1, and W2) are used to create different displays. The formula uses a modified Fourier series to sum the initial random X and Y values to produce a newX, newY. The program for this is:

```
LOOP
  [illegible]
[illegible]
  [illegible]
[illegible]
  newX=(X+W1+E*P1/PP) MOD 1
  newY=(Y+W2+E*P2/PP) MOD 1
  GOTO LOOP
```

The A and B variables are predefined in the program, PP is 2*PI and E, W1, and W2 are the three variables that produce the display. MOD means to divide and use the remainder, so MOD 1 eliminates the whole number portion and uses only the decimal. The points actually plotted are (639*X,399*Y). There are twelve Chaos data lines at the end of the listing.

When you start the program, CHAOS, there is a reminder of what the hot keys are. Using the RMB you'll see that there are two groups of displays, Symmetry and Chaos. You can switch at any time to any menu item. Each drawing starts with it's own palette but you can press F1 through F10 to change palettes. And, at any time, you can press the up-arrow key to cycle the palette colors except for the background color.

## HOW TO COLOR

A key question was how to color the display. Using the distance from a fixed point produced circular bands of color and changing color every other set produced real chaos. I finally decided to color a point depending on how many times it's been PSET. This requires an array for each pixel on the screen – all 639*399 of them! Every time a point on the screen is set the same location in the array is increased by 1. The value in the array is used to look up the corresponding value in COLORSCALE, an array of colors. If a point has been set 102 times, the 102nd value in COLORSCALE is 5, so that color is used to PSET that point. The maximum number of times a point is set is 255. While this doesn't seem like very many times, it takes quite a while for the entire display to become colorful. By the way, some palettes start with very low colors so it may look like a blank screen, but don't worry, they're on their way. I would let each display run for at least 10 minutes or more; to get the full effect let each display run for about an hour. While the picture is drawing try all the palettes and color cycle each. Oh, the names for the displays are entirely my own and reflect the author's opinion.

Now let's look at the program, Listing 1. Notice that my include files are in quotes. All the variables for both halves of the listing, Symmetry and Chaos, are equated or listed. The actual A and B values for Chaos are shown, but the B values are multiplied by PP (2*PI) before storing them at the end of the program. Look at the P1 and P2 formulas again and you'll see why that's much quicker. After opening a 640x400 screen and window, the libraries and a 640x400 array the program reminds you of the hot keys and waits for a menu selection. Notice that for the Symmetry portion variable V1 is stored in a5 since it's followed by all the Symmetry variables. In Chaos variable V2 is stored in a5. The starting X and Y location can be changed for either program but will eventually produce the same pattern but not necessarily in the same order. You'll notice that some of the patterns have a few dots in areas that never get set again. This is because most Chaos equations take up to 100 iterations before they finally settle down into a pattern. Rather than just compute 100 blank points, I went ahead and PSET all of them.

CONVERTDP is used by both programs to convert the variables to double-precision numbers. Since XSCALE and YSCALE for Chaos are always 639 and 399, their dp values have been precomputed and stored in the V2 variable array. Also, E gets divided by PP before it's value is saved since that would have to be done twice during every newX and newY computation. And, X and Y are also multiplied by PP but at the start of each computation rather than doing that eight different times during the computations.

After newX and newY coordinates have been computed the color is looked up and that location is PSET. Then a menu check is made to see if you want to change palettes, color cycle, pick a new menu item, or quit. If you choose a new item the program first checks what menu it's in and then branches to the item check for that menu. To color cycle, all palettes go into a general location called COLORTABLE as you use them. When you press the up-arrow key color1 is stored in d2 then each color value decreases by 1; at the end, color1 is restored as color15.

## SYMMETRY

Now let's discuss the actual computations in each program. Both free the memory array first since there may be values in there from a previous display. This is immediately followed by a new array call. This seemed easier and quicker than clearing the entire array to 0. When you start either program, fp7 contains the initial X location, fp6 contains the initial Y location, a5 is the variable array address, a4 the COLORSCALE address, and a3 the COLORS array address.

In Symmetry, X is moved to fp0 and Y moved to fp2. Each is multiplied by itself, added together and stored in fp2 as ZZBAR (X*X+Y*Y). X is then moved to fp5 as ZREAL and Y to fp4 as ZIMAG. You can already begin to see the advantage of having eight more registers and the ability to perform any operation on any combination of registers. Also, having eight registers lets us leave some often used values in a register rather than calling them each time they're used.

The degree of rotation (N) is stored in d0 as a counter. Next, Y and ZIMAG are multiplied, X and ZREAL multiplied, subtracted from the previous operation and saved in fp3 as ZA. Then X and ZIMAG are multiplied. Y and ZREAL are multiplied replacing the old ZREAL and ZIMAG. Fp5 is saved as the new ZIMAG and ZA as the new ZREAL. This repeats until d0 decreases to below 0.

Now multiply Y times ZIMAG, X times ZREAL, and subtract them. Multiply this by variable B using its location as B(a5); multiply ZZBAR (in fp2) by A using A(a5) and finally, add L(a5). This is the temporary value P in fp3. To get the new X location, multiply G times ZREAL, O times Y, and subtract; then add P times X. Fp5 now contains the new X coordinate. At this point, fp0 contains G, fp2 contains O, and fp3 contains P. We can then directly multiply O times X, G times ZREAL, and P times Y. Fp6 will contain the new Y and then move fp5 to fp7 as the new X. By planning ahead, all the variables can be used in most of the registers at least once.

To compute the points that will be PSET, multiply X by XSCALE and store the result in ACROSS, adding 320 to center it. Multiply Y by YSCALE and store the result in both d0 and DOWN adding 200 to center it. The color to be used is in the 640X400 array COLORS (in a2). Multiply the down location (in d0) by 640 and add ACROSS. Add this distance to the value in a4 and store the contents of that location in COUNT - MOVE.B 0(A3,D0.L),COUNT. Add 1 to COUNT; if the result is 0, we've already set that point 255 times so skip directly to computing the next point. If it's not 0, replace the increased COUNT back to the array - MOVE.B COUNT,0(A3,D0.L). Now get the COUNT value from the COLORSCALE array in a4 - MOVE.B 0(A4,COUNT),D0. Remember that COUNT was equated to register d4 back at the beginning of the program. Use the color value in d0 to PSET the point.

This is followed by a message check. Since one of my IDCMP flags was RAWKEY, the routine looks to see if you have pressed F1 through F10 (rawkey #$50 - #$59) or the up-arrow (#$4C). If you have, the program installs a new palette and continues drawing or cycles the colors by 1. RAWKEY values are different from VANILLAKEY values. While the later is usually the ASCII value for a character, RAWKEY values are based more on that key's placement and location on the keyboard; using RAWKEY lets you test for any key or combination of keys as "hot keys". When you choose a menu, the program goes all the way back to HANDLE_MENU0 or HANDLE_MENU1. If you don't do anything the display keeps drawing.

## CHAOS

The Chaos program initially frees, then resets the COLORS array, puts XSTART in fp4, YSTART in fp5, PP in fp6, and #1 in fp2. The array locations are VS in a5, COLORSCALE in a4, and COLORS in a3. Then E is moved to fp0, divided by PP, and stored back in E. This is because the Chaos formulas both use 2 times a value divided PP. After clearing the screen the actual computations begin.

First, multiply X and Y by PP and save their product in fp4 (PPX) and fp3 (PPY). Move PPX to fp0 and add B11(a5). Remember that B11 has already been multiplied by PP. Get the sine of this value and then multiply by A11(a5); keep this result (A11*SIN(PP*X+B11)) in fp0. Now, add PPX and B12, get the sine, multiply by A12, and add this to the

DJComputers.cz

## TABLE I
## FP Commands

**SINGLE REGISTER**

| Command | Description |
|---|---|
| FABS | - absolute value |
| FACOS | - arc cosine |
| FASIN | - arc sine |
| FATAN | - arc tangent |
| FATANH | - arc tangent hyperbolic |
| FCOS | - cosine |
| FCOSH | - cosine hyperbolic |
| FETOX | - E to the X power |
| FETOXM1 | - E to the X-1 power |
| FGETEXP | - get exponent |
| FGETMAN | - get mantissa |
| FINT | - integer value |
| FLOG10 | - log, base 10 |
| FLOG2 | - log, base 2 |
| FLOGN | - log, base E |
| FLOGNP1 | - log, base E of X+1 |
| FNEG | - negate |
| FSIN | - sine |
| FSINH | - sine hyperbolic |
| FSQRT | - square root |
| FTAN | - tangent |
| FTANH | - tangent hyperbolic |
| FTENTOX | - 10 to the X power |
| FTWOTOX | - 2 to the X power |

**MULTIPLE REGISTERS**

| Command | Description |
|---|---|
| FADD | - add registers |
| FCMP | - compare registers |
| FDIV | - divide registers |
| FMOD | - modulo remainder |
| FMUL | - multiply registers |
| FSUB | - subtract registers |

**SPECIAL COMMANDS**

| Command | Description |
|---|---|
| FBxx | - branch conditionally |
| FDBxx | - decrease register, branch |
| FMOVE | - move data |
| FNOP | - no operation |
| FSxx | - set according to condition |
| FSINCOS | - get both sine and cosine |
| FTST | - test for 0, positive, or negative |

previous result to fp1. Next, add PPX, PPY, and B13, get the sine, multiply by A13, and add to fp0. Finally, subtract PPY from PPX, add B14, get the sine, multiply by A14, and add to fp1. Multiply fp0 times E (actually E/PP), add W1, and add this to the origin X in fp7. We need to keep this value below 1 so FMOD with fp2 (containing 1); this will delete the whole number and use just the decimal. MOD, by the way, uses only the result after dividing.

Follow the same procedure to compute a new Y, add this to the previous Y value, and FMOD it. The points to be PSET are X*XSCALE and Y*YSCALE. The dp scales of 639 and 399 were previously computed. You don't need to check screen values in this case since X and Y can't reach or exceed 1 and the scales are the maximum screen values. If you add a zoom routine, however, you need the boundary checks since much of the display would be off the screen and exceed the values in the COLORS array. Obtaining the color values is the same as for Symmetry as is the message check.

## CHANGES

An obvious addition to this program would be an extra item for each menu that allows you to choose your own variables for either program. This would probably have to be done with requesters. Keep the scales small (128,128) to begin with in the Symmetry program. Weird things happen when you try to PSET a point off the screen in assembly language. And you'll also be selecting an incorrect value in the color array. The Chaos portion could easily include a zoom routine. Many displays, especially Lava Flow, look much better in close up.

Assemble this program using PASM as CHAOS. If you want to try the PHXASS options, assemble it using PHXASS CHAOS.ASM options and link it using PHXLNK CHAOS.O; you can delete the CHAOS.O file later. Run the program as CHAOS. Listing1, Chaos, the include files, and PHX files with their docs are included on the magazine disk. I've also included some pictures made using this program.

I hope you've enjoyed learning about the math coprocessor and it's new commands; they'll be used in future articles along with the PHXASS assembler. Remember, when things get dull, put a little CHAOS in your life.

## Listing One

```
;   CHAOS
    opt   l                        ; [illegible]
    section [illegible],code
    bra   main
    include "[illegible]"
    include "[illegible]"
    include "[illegible]"
    include "[illegible]"
equates:
[illegible]             equ 1
[illegible]             equ 2
[illegible]             equ 3
size                    equ $[illegible]    ;[illegible]
```

```
offsets:
[illegible]         equ -354
loadrgb4            equ -192
;a0-vp,a1-colortable,d0-items
setapen             equ -342
writepixel          equ -324
[illegible]         equ  234
move                equ  240
text                equ  60
[illegible]   equ   65
down          equ   65
count         equ   64

;Symmetry variables
;   fp7   -   x
;   fp6   -   y
;   fp5   -   zreal,zexx
;   fp4   -   zimag,zexy
;   fp3   -   za,p
;   fp2   -   temp,c
;   fp1   -   temp
;   fp0   -   temp,q

l     equ    0
r     equ    8
[illegible]    equ    16
[illegible]    equ    24
[illegible]    equ    32
xscale      equ     44
yscale      equ     52

;Chaos variables
;   fp7   -   x
;   fp6   -   y
;   fp5   -   sin(2*pi)
;   fp4   -   pp*x
;   fp3   -   pp*y
;   fp2   -   1
;   fp1   -   temp
;   fp0   -   temp

[illegible]     equ     0
w1              equ     8
w2              equ     16
xscale1         equ     24
yscale1         equ     32
a11       equ     40
a12       equ     48
a13       equ     56
a14       equ     64
b11       equ     72
b12       equ     80
b13       equ     88
b14       equ     96
a21       equ     104
a22       equ     112
a22       equ     120
a24       equ     128
b21       equ     136
b22       equ     144
```



Downloaded from amigaguide? zz sieheupd.org

```
b23     equ     152
b24     equ     100

;a11dp equ.d -.2681366364875 4
;a12dp equ.d  .9106549935679 0
;a13dp equ.d  .3111720763927 53
;a14dp equ.d  .0400987763547 0
;b11dp equ.d  .8954006429660 5
;b12dp equ.d  .5344004360999 1
;b13dp equ.d  .9470947542307 8
;a14dp equ.d  .2335601555085 07
;a21dp equ.d  .0381861167151 2
;a22dp equ.d -.5950290590046 0
;a23dp equ.d  .1529955487278 6
;a24dp equ.d -.8015986197615
;b21dp equ.d  .4960722865603 9
;b22dp equ.d  .3362859701275 8
;b23dp equ.d  .2080490173097 1
;b24dp equ.d  .1550467277777 7



macros
gfxlib  macro               ;(routine)
  movea.l gfxbase(pc),a6
  jsr     \1(a6)
  endm
point macro
  move.l  rp(pc),a1
  move.w  ac+xee,d0
  move.w  #300,d1
  sub.w   down,d1
  gfxlib  writepixel
  endm
color macro
  move.l  color(pc),a1
  gfxlib  setapen
  endm
palette macro
  movea.l vp(pc),a0   ;(colortable)
  lea     \1(pc),a1
  moveq   #15,d0
  gfxlib  loadrgb4
  endm
newpalette macro      ;(colortable)
  lea     \1(pc),a1
  lea     colortable(pc),a0
  move.w  #7,d0       ;8 long word values
swap\@
  move.l  (a1)+,(a0)+
  dbra.s  d0,swap\@
  palette colortable ;use the new palette
  endm
pale    macro
  moveq   #0,d0
  movea.l rp(pc),a1
  gfxlib  setrast
  endm
qprint  macro  x,y,color,msg,length
  movea.l rp(pc),a1
  move.w  #\1,d0
  move.w  #\2,d1
  gfxlib  move
  moveq   #\3,d0
  color
  movea.l rp(pc),a1
  lea     \4(pc),a0
  moveq   #\5,d0
  gfxlib  text
  endm

main:
  move.l  sp,stack
openlibs:
  openlib int,done
  openlib gfx,close_int
  openlib mathtt,close_int
set_up:
make_screen:
  openscreen myscreen,close_libs
  openwindow mywindow,close_screen
  openmenu menu0
  movea.l rp(pc),a1
  move.l  #jam1,d0
  gfxlib  setdrmd
memory:
  array   colors,close_window
  qprint  135,180,2,11msg,47
  qprint  135,200,2,c6msg,40
  qprint  135,250,1,cmmsg,25
msg_check
  gfx     msg_check
  cmp.l   #menupick,d2
  beq.s   check_menus
  bra.s   msg_check
check_menus
  eval_menunumber        ;d0=menu#,d1=item#
  tst.w   d0
  beq.s   handle_menu0   ;symmetry
  cmpi.w  #1,d0
  beq     handle_menu1   ;chaos
  bra.s   msg_check
handle_menu0
  cmpi.w  #0,d1
  beq.s   do_data0
  cmpi.w  #1,d1
  beq     do_data1
  cmpi.w  #2,d1
  beq     do_data2
  cmpi.w  #3,d1
  beq     do_data3
  cmpi.w  #4,d1
  beq     do_data4
  cmpi.w  #5,d1
  beq     do_data5
  cmpi.w  #6,d1
  beq     do_data6
  cmpi.w  #7,d1
  beq     do_data7
  cmpi.w  #8,d1
  beq     do_data8
```

DJComputers.cz

```
    cmpi.w    #9,d1
    beq       do_data9
    cmpi.w    #10,d1
    beq       do_data10
    cmpi.w    #11,d1
    beq       do_data11
    cmpi.w    #12,d1
    beq       quit
    bra       mnu_check
do_data0                        ;autumn
    newpalette  colortable4
    lea       data0(pc),a0
    bra       get_values
do_data1                        ;explico
    newpalette  colortable5
    lea       data1(pc),a0
    bra       get_values
do_data2                        ;chinese plate
    newpalette  colortable1
    lea       data2(pc),a0
    bra       get_values
do_data3                        ;ancient q ...
    newpalette  colortable5
    lea       data3(pc),a0
    bra       get_values
do_data4                        ;origami
    newpalette  colortable7
    lea       data4(pc),a0
    bra       get_values
do_data5                        ;lariats
    newpalette  colortable8
    lea       data5(pc),a0
    bra       get_values
do_data6                        ;civil defense
    newpalette  colortable9
    lea       data6(pc),a0
    bra       get_values
do_data7                        ;pentagram
    newpalette  colortable3
    lea       data7(pc),a0
    bra       get_values
do_data8                        ;sparkle
    newpalette  colortable6
    lea       data8(pc),a0
    bra       get_values
do_data9                        ;three fold
    newpalette  colortable6
    lea       data9(pc),a0
    bra.s     get_values
do_data10                       ;rings
    newpalette  colortable5
    lea       data10(pc),a0
    bra.s     get_values
do_data11                       ;persian
    newpalette  colortable1
    lea       data11(pc),a0
get_values:
    jsr       convertdp
    movedp    lox

    movea.l   a5,a0
    jsr       convertdp
    movedp    hix

    movea.l   a5,a0
    jsr       convertdp
    movedp    ldp

    movea.l   a5,a0
    bsr       convertdp
    movedp    qcx

    movea.l   a5,a0
    jsr       convertdp
    movedp    sdp

    movea.l   a5,a0
    jsr       convertdp
   movea.l  dpmathbase(pc),a6
    jsr       -$1e(a6)          ;dpfix
    subq.l    #3,d0             ;use 3 - 0
    move.l    d0,c

    movea.l   a5,a0
    bsr       convertdp
    movedp    xscaledp

    movea.l   a5,a0
    jsr       convertdp
    movedp    yscaledp

    free      colors            ;clear previous values
    display   colors,close window  ;start anew
    fmove.d   #-2.015625,fp7        ;x start
    fmove.d   #-1.0039375,fp6       ;y start
   movea.l  colors(pc),a3
    lea       colorscale(pc),a4
    lea       vl(pc),a1
    pdlz
    bra       continue1

handle_menu1
    cmpi.w    #0,d1
    beq.s     do_chaos0
    cmpi.w    #1,d1
    beq       do_chaos1
    cmpi.w    #2,d1
    beq       do_chaos2
    cmpi.w    #3,d1
    beq       do_chaos3
    cmpi.w    #4,d1
    beq       do_chaos4
    cmpi.w    #5,d1
    beq       do_chaos5
    cmpi.w    #6,d1
    beq       do_chaos6
    cmpi.w    #7,d1
    beq       do_chaos7
    cmpi.w    #8,d1
    beq       do_chaos8
```

pdcomputers.cz

```
    cmpi.w   #9,d1
    beq      do_chaos9
    cmpi.w   #10,d1
    beq      do_chaos10
    cmpi.w   #11,d1
    beq      do_chaos11
    cmpi.w   #12,d1
    beq      quit
    bra      msg_check
do_chaos0                   ;sand dune
   newpalette colortable2
    lea      chaos0(pc),a0
    bra      get_values1
do_chaos1                   ;lava flow
   newpalette colortable3
    lea      chaos1(pc),a0
    bra      get_values1
do_chaos2                   ;swirls
   newpalette colortable1
    lea      chaos2(pc),a0
    bra      get_values1
do_chaos3                   ;[illegible] wall
   newpalette colortable0
    lea      chaos3(pc),a0
    bra      get_values1
do_chaos4                   ;dragon tail
   newpalette colortable4
    lea      chaos4(pc),a0
    bra      get_values1
do_chaos5                   ;[illegible] floor
   newpalette colortable6
    lea      chaos5(pc),a0
    bra      get_values1
do_chaos6                   ;waves
   newpalette colortable7
    lea      chaos6(pc),a0
    bra      get_values1
do_chaos7                   ;birds on a [illegible]
   newpalette colortable5
    lea      chaos7(pc),a0
    bra      get_values1
do_chaos8                   ;swaying flowers
   newpalette colortable8
    lea      chaos8(pc),a0
    bra      get_values1
do_chaos9                   ;barbed wire
   newpalette colortable1
    lea      chaos9(pc),a0
    bra.s    get_values1
do_chaos10                  ;corona
   newpalette colortable9
    lea      chaos10(pc),a0
    bra.s    get_values1
do_chaos11                  ;profile
   newpalette colortable10
    lea      chaos11(pc),a0
get_values1:
    bsr      convertdp
    movedp   [illegible]
    moveq.l  #5,d0
    bsr      convertdp
    movedp   width

    movea.l  a1,a0
    bsr      convertdp
    movedp   [illegible]

    free     colors
    array    colors,close_window
    fmove.d  #[illegible],fp7      ;start x
    fmove.d  #[illegible],fp6      ;start y
    fmove.d  #6.283185307179586,fp5  ;2*pi
    fmove.l  #1,fp2
    lea      [illegible](pc),a5
    lea      colorscale(pc),a4
    movea.l  colors(pc),a3
    fmove.d  [illegible](a5),fp0
    fdiv.x   fp5,fp0
    fmove.x  fp0,[illegible]      ;c=2/(2*pi)
    [illegible]
    bra      continue2

convertdp
    moveq.l  #0,d0
    moveq.l  #0,d1
    moveq.l  #0,d4
    moveq    #0,d5
    suba.l   a2,a2          ;[illegible]
    cmpi.b   #'-',(a0)      ;negative ?
    bne.s    positive
    bset     #31,d7         ;set negative sign bit
    addq.l   #1,a0          ;next ?
positive
getadigit
    move.b   (a0)+,d5
    cmpi.b   #'.',d5        ;decimal ?
    bne.s    testdigit
    move.w   #1,a2          ;decimal flag
    moveq    #0,d7
    bra.s    getadigit
testdigit
    cmpi.b   #'9',d5
    bhi.s    zerocheck
    cmpi.b   #'0',d5
    blt.s    zerocheck
    andi.l   #$0f,d5
    move.l   d1,d2
    move.l   d1,d3
    asl.l    #1,d1
    roxl.l   #1,d0
    asl.l    #1,d3
    roxl.l   #1,d2
    asl.l    #1,d3
    roxl.l   #1,d2
    asl.l    #1,d3
    roxl.l   #1,d2
    moveq.l  #0,d6
    add.l    d3,d1
    addx.l   d2,d0    ;d0 = d0 * 10
```

```
    add.l    d5,d1
    addx.l   d6,d0     ;d0 = d0 * 10 + digit
    addq.w   #1,d7
    cmpi.w   #16,d7    ;get up to 16 digits
    bne.s    getadigit
zerocheck
    move.l   d0,d5
    tst.l    d1
    bne.s    1$
    tst.l    d0
    beq.s    dp_done
1$
   move.l   #$433,d6 ;maximum exponent
2$
    subq.l   #1,d6
    asl.l    #1,d1
    roxl.l   #1,d0
    bcc.s    2$
    moveq.l  #11,d5    ;# times to shift
shiftdown
    lsr.l    #1,d0
    roxr.l   #1,d1
    dbra     d5,shiftdown
    swap     d6        ;exponent to high bits
    asl.l    #4,d6
    or.l     d6,d0
    cmpi     #0,d7     ;any decimal
    beq.s    do_sign
    subq.l   #1,d7     ;any more digits ?
    bmi.s    do_sign
fractionalize
    move.l   #$40240000,d2
    moveq.l  #0,d3     ;10dp
   movea.l  dmathbase(pc),a6
    jsr      -84(a6)   ;dpdiv
    dbra     d7,fractionalize
do_sign
    or.l     d4,d0
dp_done
    moveq.l  #0,d6     ;optional OK check
    rts

;symmetry program
continue
     fmove.x  fp7,fp0      ;x
     fmove.x  fp6,fp2      ;y
     fmul.x   fp0,fp0      ;x * x
     fmul.x   fp2,fp2      ;y * y
     fadd.x   fp0,fp2      ;zzbar

     fmove.x  fp7,fp5      ;x -> zreal
     fmove.x  fp6,fp4      ;y -> zimaginary
     move.l   a(pc),d3

iterate
     fmove.x  fp6,fp0      ;y
     fmul.x   fp4,fp0      ;y * zi
     fmove.x  fp7,fp1      ;x
     fmul.x   fp5,fp1      ;x * zr
     fsub.x   fp0,fp1      ;(x * zr) - (y * zi)
     fmove.x  fp1,fp3      ;xn

     fmul.x   fp7,fp4      ;x * zi
     fmul.x   fp6,fp5      ;y * zr
     fadd.x   fp5,fp4      ;new zimaginary
     fmove.x  fp1,fp5      ;new zreal
     dbra     d0,iterate

getp       ;((x * zr) - (y * zi)) * b - (a * zzbar) + l
     fmove.x  fp6,fp0      ;y
     fmul.x   fp4,fp0      ;y * zi
     fmove.x  fp7,fp1      ;x
     fmul.x   fp5,fp1      ;x * zr
     fsub.x   fp0,fp1      ;(x * zr) - (y * zi)
     fmul.d   b(a5),fp1
     fmul.d   a(a5),fp2
     fadd.x   fp2,fp1      ;(a * zzbar) - b((x * zr) - (y *
zi))
    fmove.d  l(a5),fp3
     fadd.x   fp1,fp3      ;p

get_new_x  ;(g * zr) - (p * y) + (p * x)
     fmove.d  g(a5),fp0    ;g
     fmul.x   fp0,fp5      ;g * zr
     fmove.x  fp6,fp1      ;y
     fmove.d  p(a5),fp2    ;p
     fmul.x   fp2,fp1      ;p * y
     fsub.x   fp1,fp5      ;(g * zr) - (p * y)
     fmove.x  fp7,fp1      ;x
     fmul.x   fp3,fp1      ;x * p
     fadd.x   fp1,fp5      ;newx

get_new_y  ;(p * x) - (g * zi) + (p * y)
     fmul.x   fp2,fp7      ;p * x
     fmul.x   fp0,fp4      ;g * zi
     fmul.x   fp3,fp6      ;p * y
     fadd.x   fp7,fp6      ;(p * x) + (p * y)
     fsub.x   fp4,fp6      ;newy
     fmove.x  fp5,fp7      ;newx

xpoint
    fmove.d  xscale(a5),fp0 ;xscale
     fmul.x   fp7,fp0      ;x * xscale
    fmove.l  fp0,across   ;int(x * xscale)
    addi.l   #320,across  ;center
ypoint
    fmove.d  yscale(a5),fp0 ;yscale
     fmul.x   fp6,fp0      ;y * yscale
     fmove.l  fp0,d0       ;int( y * yscale)
     addi.l   #200,d0
     move.l   d0,down

     mulu     #640,d0
     add.l    across,d0
     moveq    #0,count
    move.b   0(a0,d0.l),count
     addq.b   #1,count
     beq.s    max_check
plotit
    move.b   count,0(a3,d0.l)
```

```
    moveq.l   #0,d0
    move.b    6(a4),[illegible]d0
    [illegible]
    [illegible]
[illegible]check
    [illegible]
    cmpi.l    #menupick,d3
    beq       check_menus
    btst.w    [illegible]
    bne       continue1
    andi.w    #$0111,d3
    cmpi.w    #$50,d3           ;palette 1
    beq.s     dol_palette1
    cmpi.w    #$51,d3
    beq.s     dol_palette2
    cmpi.w    #$52,d3
    beq       dol_palette3
    cmpi.w    #$53,d3
    beq       dol_palette4
    cmpi.w    #$54,d3
    beq       dol_palette5
    cmpi.w    #$55,d3
    beq       dol_palette6
    cmpi.w    #$56,d3
    beq       dol_palette7
    cmpi.w    #$57,d3
    beq       dol_palette8
    cmpi.w    #$58,d3
    beq       dol_palette9
    cmpi.w    #$59,d3
    beq       dol_palette0
    cmpi.w    #$40,d3
    beq       colorcycle1
    bra       continue1
dol_palette1
    newpalette colortable1
    bra       continue1
dol_palette2
    newpalette colortable2
    bra       continue1
dol_palette3
    newpalette colortable3
    bra       continue1
dol_palette4
    newpalette colortable4
    bra       continue1
dol_palette5
    newpalette colortable5
    bra       continue1
dol_palette6
    newpalette colortable6
    bra       continue1
dol_palette7
    newpalette colortable7
    bra       continue1
dol_palette8
    newpalette colortable8
    bra       continue1
dol_palette9
    newpalette colortable9
    bra       continue1
dol_palette0
    newpalette colortable0
    bra       continue1
colorcycle1
    lea       colortable1+2,a0
    move.w    (a0),d2           ;save #1
    move.w    #1,d1
    move.w    #14,d0            ;#s 0-14 changed
cycle1
    move.w    2(a0,d1.w),0(a0,d1.w)   ;move up one
    addq.w    #2,d1             ;next color
    dbra      d0,cycle1
    move.w    d2,30(a0)         ;new #15
setcolors1
    palette   colortable
    bra       continue1

check_menus1
    [illegible]
    tst.w     d0
    beq       handle_menu0
    cmpi.w    #1,d0
    beq       handle_menu1
    bra       continue1

;[illegible]
continue2
    fmove.x   fp2,fp4      ;x
    fmul.x    fp5,fp4      ;x * 2pi
    fmove.x   fp6,fp3      ;y
    fmul.x    fp5,fp3      ;y * 2pi
get_pixel2
    fmove.x   fp4,fp0      ;ppx
    fadd.d    [illegible]  ;[illegible]
    [illegible]
    [illegible]
    fmove.x   fp3,fp1      ;ppy
    fadd.d    [illegible]  ;ppy + [illegible]
    [illegible]
    [illegible]
    fadd.x    fp1,fp0
    fmove.x   fp0,fp1      ;ppx
    fadd.x    fp3,fp1      ;ppx + ppy
    [illegible]
    [illegible]
    [illegible]
    [illegible]
    fmove.x   [illegible]  ;ppx
    fsub.x    fp3,fp0      ;ppx - ppy
    [illegible]
    [illegible]
    [illegible]
    fadd.x    fp1,fp0
    [illegible]
    [illegible]
    fadd.x    fp5,fp7      ;[illegible]
    [illegible]
```

```
qet_newy1
    fmove.x  fp4,fp0        ;oldy
    fmul.d   [illegible](a5),fp0 ;ppx + [illegible]
    fsin.x   fp0            ;sin(ppx + [illegible])
    fmul.d   [illegible](a5),fp0 ;a21 * sin(ppx + b21)
    fmove.x  fp7,fp1        ;ppy
    fmul.d   b22(a5),fp1    ;ppy + b22
    fsin.x   fp1            ;sin(ppy + b22)
    fmul.d   a22(a5),fp1    ;a22 * sin(ppy + b22)
    fadd.x   fp1,fp0
    fmove.x  fp4,fp1        ;ppx
    fadd.x   fp7,fp1        ;ppx + ppy
    fadd.d   [illegible](a5),fp1 ;ppx + ppy + b23
    fsin.x   fp1            ;sin(ppx + ppy + b23)
    fmul.d   a23(a5),fp1    ;a23 * sin(ppx +ppy +b23)
    fadd.x   fp1,fp0
    fmove.x  fp4,fp1        ;ppx
    fsub.x   fp7,fp1        ;ppx - ppy
    fadd.d   b24(a5),fp1    ;ppx - ppy + b24
    fsin.x   fp1            ;sin(ppx - ppy + b24)
    fmul.d   a24(a5),fp1    ;a24 * sin(ppx - ppy + b24)
    fadd.x   fp1,fp0
    fmul.d   e(a5),fp0      ;e * expr
    fadd.d   y2(a5),fp0     ; + y2
    fadd.x   fp3,fp0        ;newy
    fmove.x  fp0,fp3        ;keep y + 1

xscale1
    fmove.d  xscale(a5),fp0 ;xscale
    fmul.x   fp7,fp0        ;x * xscale
    fmove.l  fp0,xcoord     ;int(x * xscale)
;   cmpi.w   #$[illegible],xcoord  ;equal to or [illegible] memory check
;   bhs.s    neg_check2
yscale1
    fmove.d  yscale(a5),fp0 ;yscale
    fmul.x   fp6,fp0        ;y * yscale
    fmove.l  fp0,d0         ;int(y * yscale)
    move.l   d0,[illegible]
;   cmpi.w   #400,[illegible]   ;equal to or [illegible] memory check
;   bhs.s    neg_check2

    mulu     #640,d0
    add.l    xcoord,d0
    move.l   d0,count
    move.b   ([illegible]),count
    [illegible]   #1,count
    beq.s    neg_check2
[illegible]
    move.b   count(a2,d0.l)
    move.l   [illegible],d0
    move.b   [illegible],count(d0)
    [illegible]
    [illegible]

neg_check2
    [illegible]   continue2
    cmp.l    [illegible]
    [illegible]   [illegible]
    [illegible]   #3
```

```
    beq      continue2
    andi.w   #$[illegible],d3
    cmpi.w   #$[illegible],d3       ;palette ?
    beq.s    do2_palette1
    cmpi.w   #$[illegible],d3
    beq.s    do2_palette2
    cmpi.w   #$[illegible],d3
    beq      do2_palette3
    cmpi.w   #$[illegible],d3
    beq      do2_palette4
    cmpi.w   #$[illegible],d3
    beq      do2_palette5
    cmpi.w   #$[illegible],d3
    beq      do2_palette6
    cmpi.w   #$[illegible],d3
    beq      do2_palette7
    cmpi.w   #$[illegible],d3
    beq      do2_palette8
    cmpi.w   #$56,d3
    beq      do2_palette9
    cmpi.w   #$[illegible],d3
    beq      do2_palette10
    cmpi.w   #$[illegible],d3
    beq      do2_palette2
    bra      continue2
do2_palette1
    newpalette colortable1
    bra      continue2
do2_palette2
    newpalette colortable2
    bra      continue2
do2_palette3
    newpalette colortable3
    bra      continue2
do2_palette4
    newpalette colortable4
    bra      continue2
do2_palette5
    newpalette colortable5
    bra      continue2
do2_palette6
    newpalette colortable6
    bra      continue2
do2_palette7
    newpalette colortable7
    bra      continue2
do2_palette8
    newpalette colortable8
    bra      continue2
do2_palette9
    newpalette colortable9
    bra      continue2
do2_palette10
    newpalette colortable10
    bra      continue2
colorcycle1
    lea      colortable(pc),a0
    move.w   2(a0),d2
    move.w   #[illegible],d1
    move.w   #1,d0
```

```
myn [illegible]
  move.w  [illegible]
  [illegible].w  #2,d1
  [illegible]  d0,[illegible]
  move.w  d1,2(a0)
[illegible]colors2
  [illegible]  colortable
  bra     continue3

check_menu2
  eval_menunumber
  move.w  d0
  bsr     handle_menu
  move.w  #1,d0
  bsr     handle_menu1
  bra     continue3

quit
free_memory
  free    colors
close_window:
  [illegible]
  closewindow
  [illegible]
  closescreen
close_libs:
  closelib gfx
close_int:
  closelib int
close_dmath
  closelib dmath
done:
  move.l  stack,[illegible]
  rts

  even
stack     dc.l  0
gfxbase   dc.l  0
intbase   dc.l  0
dmathbase dc.l  0
colors    dc.l  0

v1
[illegible]dp      dc.l  0,0
adp       dc.l  0,0
bdp       dc.l  0,0
gdp       dc.l  0,0
cdp       dc.l  0,0
n         dc.l  0
xscalexp  dc.l  0,0
yscalexp  dc.l  0,0

v2
adp       dc.l  0,0
[illegible]dp      dc.l  0,0
[illegible]dp      dc.l  0,0
      dc.l  $40[illegible],0          ;xscale-[illegible]
      dc.l  $40[illegible],0          ;yscale [illegible]
      dc.l  $bfd[illegible],$[illegible]  ;a1[illegible]
      dc.l  $bf[illegible],$[illegible]  ;a1[illegible]
      dc.l  $[illegible],$[illegible]  ;a1[illegible]
      dc.l  $[illegible],$[illegible]  ;a2[illegible]
      dc.l  $4010[illegible],$c4b5d[illegible]  ;b11
      dc.l  $400[illegible],$[illegible]  ;b12
      dc.l  $401[illegible],$[illegible]44f[illegible]  ;b13
      dc.l  $3f[illegible],$[illegible]  ;b14
      dc.l  $[illegible],$[illegible]  ;a21
      dc.l  $[illegible],$[illegible]  ;a22
      dc.l  $[illegible],$[illegible]  ;a23
      dc.l  $bf[illegible],$c2e[illegible]  ;a24
      dc.l  $401[illegible],$b[illegible]  ;b21
      dc.l  $400[illegible],$[illegible]  ;b22
      dc.l  $[illegible],$[illegible]  ;b23
      dc.l  $[illegible],$[illegible]  ;b24
  even
gfx      dc.b  'graphics.library',0
  even
int      dc.b  'intuition.library',0
  even
dmath    dc.b  'mathieeedoubbas.library',0
  even
[illegible]  dc.b  'Press [illegible]  [illegible] while
drawing',0
  even
[illegible]  dc.b  '        Press the [illegible]-arrow to color
cycle',0
  even
[illegible]  dc.b  '            Now select any menu item',0
  even
myscreen
  dc.w  0,0,640,400,[illegible]
  dc.b  0,1
  dc.w  $8004
  dc.w  customscreen
  dc.l  0,0,0,0
  even
mywindow
  dc.w  0,0,640,400
  dc.b  0,1
  dc.l  [illegible]
  dc.l  [illegible]
  dc.l  0,0
  dc.l  0
  dc.l  0,0
  dc.w  0,0,0,0
  dc.w  customscreen
  even

colortable1
  dc.w  $000,$[illegible],$[illegible],$[illegible]
  dc.w  $[illegible],$[illegible],$[illegible],$[illegible]
  dc.w  $[illegible],$[illegible],$[illegible],$[illegible]
  dc.w  $[illegible],$[illegible],$[illegible],$000
  even
colortable2
  dc.w  $000,$[illegible],$[illegible],$[illegible]
  dc.w  $[illegible],$[illegible],$[illegible],$[illegible]
  dc.w  $[illegible],$[illegible],$[illegible],$[illegible]
```

```
    dc.w $c40,$f00,$ea0,$ee0
     even
colortable3
    dc.w $000,$200,$400,$600
    dc.w $800,$a00,$c00,$e00
    dc.w $e10,$e30,$e50,$e70
    dc.w $f90,$fb0,$fd0,$ff0
     even
colortable4
    dc.w $a20,$[illegible],$fd2,$fe1
    dc.w $187,$[illegible],$[illegible],$f[illegible]
    dc.w $670,$745,$870,$[illegible]
    dc.w $5a0,$5b0,$[illegible],$[illegible]
     even
colortable5
    dc.w $a[illegible],$[illegible],$710,$500
    dc.w $700,$[illegible],$800,$400
    dc.w $a40,$a[illegible],$be0,$[illegible]
    dc.w $[illegible],$d90,$[illegible],$ee0
     even
colortable6
    dc.w $004,$000,$fe6,$[illegible]
    dc.w $eb0,$eb3,$f00,$[illegible]
    dc.w $d99,$a8[illegible],$a00,$[illegible]
    dc.w $954,$8[illegible],$841,$f00
     even
colortable7
    dc.w $004,$[illegible],$0[illegible],$0[illegible]
    dc.w $0e[illegible],$[illegible],$07d,$[illegible]
    dc.w $e5c,$[illegible]e,$520,$[illegible]
    dc.w $05e,$0[illegible],$0[illegible],$[illegible]
     even
colortable8
    dc.w $[illegible],$e9b,$e00,$d[illegible]
    dc.w $c41,$d[illegible],$210,$[illegible]
    dc.w $[illegible],$[illegible],$[illegible],$[illegible]
    dc.w $1[illegible],$[illegible],$fff,$[illegible]
     even
colortable9
    dc.w $000,$0e0,$0b0,$[illegible]
    dc.w $0[illegible],$0[illegible],$05[illegible],$0[illegible]
    dc.w $0[illegible],$[illegible],$[illegible],$[illegible]
    dc.w $[illegible],$0[illegible],$0[illegible],$[illegible]
     even
colortable10
    dc.w $13[illegible],$[illegible],$[illegible],$[illegible]
    dc.w $[illegible],$[illegible],$[illegible],$[illegible]
    dc.w $[illegible],$[illegible],$[illegible],$[illegible]
    dc.w $[illegible],$[illegible],$[illegible],$[illegible]
     even
colortable
    dc.w 0,0,0,0,0,0,0,0
    dc.w 0,0,0,0,0,0,0,0
     even
colorscale
     dc.b 1,0
     dc.b 2,1
     dc.b 3,2
     dc.b 4,3
     dc.b 6,4
     dc.b 10,5
     dc.b 13,6
     dc.b 16,7
     dc.b 19,8
     dc.b 22,9
     dc.b 25,10
     dc.b 28,11
     dc.b 31,12
     dc.b 34,13
     dc.b 37,14
     dc.b 40,15
     even

menu
  makemenu menu0,'Symmetry',menu1,0,1
   makeitem menu0item0,'Autumn',menu0item1,0,$[illegible],$[illegible]
   makeitem menu0item1,'Spring',menu0item2,10,$53,$[illegible]
    makeitem menu0item2,'[illegible]
Plate',menu0item3,20,$53,$[illegible]
    makeitem menu0item3,'Stained
Glass',menu0item4,30,$53,$[illegible]
   makeitem menu0item4,'Origami',menu0item5,40,$53,$fe[illegible]
   makeitem menu0item5,'[illegible]',menu0item6,50,$53,$f[illegible]
    makeitem menu0item6,'[illegible]
[illegible]',menu0item7,60,$[illegible],$[illegible]
     makeitem
menu0item7,'[illegible]',menu0item8,70,$53,$[illegible]
   makeitem menu0item8,'[illegible]',menu0item9,80,$[illegible],$[illegible]
    makeitem menu0item9,'Three
Fold',menu0item10,90,$53,$[illegible]
     makeitem
menu0item10,'Rings',menu0item11,100,$53,$[illegible]
    makeitem menu0item11,'[illegible]
[illegible]',menu0item12,110,$53,$[illegible]
    makeitem menu0item12,'QUIT',0,120,$[illegible]
    even
  makemenu menu1,'Choose',0,100,1
    makeitem menu1item0,'Sand
Dunes',menu1item1,0,$53,$[illegible]
    makeitem menu1item1,'Lava
Flow',menu1item2,10,$53,$[illegible]
   makeitem menu1item2,'[illegible]',menu1item3,20,$53,$ffb
    makeitem menu1item3,'[illegible]
[illegible]',menu1item4,30,$53,$[illegible]
    makeitem menu1item4,'[illegible]
[illegible]',menu1item5,40,$53,$fe[illegible]
    makeitem menu1item5,'Ocean
[illegible]',menu1item6,50,$53,$[illegible]
   makeitem menu1item6,'[illegible]',menu1item7,60,$[illegible],$fb[illegible]
    makeitem menu1item7,'[illegible]
Leaves',menu1item8,70,$53,$[illegible]
    makeitem menu1item8,'Swaying
Flowers',menu1item9,80,$53,$[illegible]
    makeitem menu1item9,'Barbed
[illegible]',menu1item10,90,$53,$[illegible]
     makeitem
menu1item10,'[illegible]',menu1item11,100,$[illegible],$[illegible]
     makeitem
menu1item11,'Eyefile',menu1item12,110,$53,$[illegible]
   makeitem menu1item12,'QUIT',0,120,$53
```

```
    even
data01
  dc.b '-2.[illegible]',0 ;[illegible]
    even
data1:
  dc.b '2.[illegible],23,256,200',0 ;[illegible]
    even
data2
  dc.b '-2.00,[illegible],234,160',0 ;[illegible]
    even
data3
  dc.b '-2,05,3,[illegible],320,255',0 ;[illegible] GLASS
    even
data4
  dc.b '-[illegible],240',0 ;[illegible]
    even
data5
  dc.b '[illegible],256,200',0 ;[illegible]
    even
data6
  dc.b '1,55,-2,[illegible],224,100',0 ;[illegible]
    even
data7
  dc.b '2.5,-2,[illegible],224,160',0 ;[illegible]
    even
data8
  dc.b '[illegible],188,5,320,256',0 ;[illegible]
    even
data9
  dc.b '[illegible]5,-1,[illegible],200,160',0 ;[illegible]
    even
data10
  dc.b '[illegible],200,100',0 ;[illegible]
    even
data11
  dc.b '2.[illegible],256,200',0 ;[illegible]
[illegible]
    even

chaos2:
  dc.b '.75,.[illegible]',0 ;[illegible]
[illegible]
    even
chaos1:
  dc.b '[illegible]',0 ;[illegible]
[illegible]
    even
chaos3
  dc.b '.5,.[illegible]',0 ;[illegible]
    even
chaos5
  dc.b '.5,.[illegible]',0 ;[illegible]
[illegible]
    even
chaos4
  dc.b '[illegible]',0 ;[illegible]
[illegible]
    even
chaos5
  dc.b '.5,[illegible]',0 ;[illegible]
    even
chaos6
  dc.b '.5,.[illegible]',0 ;[illegible]
    even
chaos7
  dc.b '[illegible]' ;[illegible]
    even
chaos8
  dc.b '.75,.[illegible]',0
[illegible]
    even
chaos9
  dc.b '[illegible]' ;[illegible]
    even
chaos10
  dc.b '.5,[illegible]',0 ;[illegible]
    even
chaos11
  dc.b '.5,[illegible]',0 ;[illegible]
    even

    end
```

Complete source code & listings can be found on the AC's TECH disk.

*Please write to:*
*William P. Nee*
*c/o AC's TECH*
*P.O. Box 2140*
*Fall River, MA 02722*

# A Guide to AmigaDOS Shared Libraries

By Daniel Stenberg

The examples of this article are written in assembler and C and require some knowledge of the languages to fully understand what they are all about. They are written only to illustrate the explanations and are only parts of larger source codes. They may not be accurate and I take no responsibility for the correctness or function of the examples.

## Shared Library Overview

To be able to learn how to make a shared library, it's important to have the knowledge about what it is all about. In this article I'll take you through all steps, from the most basic ones down to the ones dealing with low level library programming.

### Shared Library

First, an answer to the question: what is a shared library? As the name says, it is a function library shared by several simultaneous tasks and processes. The shared library code is not present in the executable image on disk, but is a separate file. The shared code is not loaded together with the executable. It is loaded into memory only when a program requires it.

On Amiga, the naming convention says that a shared library always be in lowercase letters with a ".library" ending, and the directory to put them in is "LIBS:".

### Link Library

A link library is not to be mixed up with a shared library. A link library is a function library that is linked into the executable at compile time. A link library becomes a part of an executable image.

### ROM Based/Disk Based Libraries

The AmigaDOS system consists of several shared libraries, whose names you recognize: dos.library, exec.library, graphics.library, only to mention a few. These libraries won't be found in the LIBS: directory, they reside in ROM. Whether in ROM or on disk, shared libraries work and are used the same way.

### Memory Usage

As mentioned, shared libraries are loaded when a program requests, i.e. opens, it. When the program has finished using the library, it closes the library. The library remains in memory even though no process is using it, until the operating system requires the memory it occupies or is forced to remove itself by a program, such as "avail FLUSH" on the shell prompt in AmigaDOS 2.0 or later.

## Other Operating Systems

Shared libraries are not AmigaDOS specific. Such are also found under UNIX and OS/2, only to mention the most obvious and common.

## Advantages

The reasons why so many systems are using shared libraries are, among others: less disk space is used because the shared library code is not included in the executable programs, less memory is used because the shared library code is only loaded once, load time may be reduced because the shared library code may already be in memory when a program wants it, and that programs using shared libraries are very easily updated.

## Calling Shared Library Functions

We've been looking at what a shared library is, a little about how it works, and some of its advantages. Now it's time to see how a library is used and accessed.

## Address Library Functions

To be able to handle library calls we must know how to call shared library functions. I'll describe it with a small comparison to standard, non-shared functions. The most significant difference is in the way the functions are addressed. A standard function within a program is more or less an address to which the program counter is set when we want to jump to it. A shared library function is on the other hand addressed by adding a number to the address of the library's base.

When using standard function calls, the compiler or assembler arrange so that e.g. the function "getname" is associated with the particular static address in memory where the "getname" function starts. If the same "getname" function would be a shared library function, the compiler wouldn't know the actual address of it, but dynamically add a certain number (index) to the library's base address to access it.

As you see, we must know the index of the function and the library base address to be able to call a shared library function.

## Library Base

To find out the library base of a shared library, you must call OpenLibrary() which will return the library base of the specified library in register D0. All library bases are found like that except exec.library's, which is found by reading the pointer stored at the absolute address 4.

## Index

Whenever you want to call a function in a shared library you (or the compiler) have to know the index to add to the library's base address.

E.g., to call OpenLibrary() you must know the index of the function and the library base itself (OpenLibrary() is an exec.library function and we know that exec.library's base address is found at address 4). A call to OpenLibrary() could look like this in assembler:

```
MOVE.L
4,a6
; SysBase is the base of
; exec.library's base address
; >>> Parameters left out in this example <<<
JSR
-552(a6)
; -552 is a negative offset into the jump
; table at the correct index. The index
; is -552 in this case.
```

## Parameters

OK, we know how to call a library function and we know that we must call OpenLibrary() to get a library's base address. To inform e.g. OpenLibrary() which library we want to open, we must send it some parameters. The documentation tells us that OpenLibrary() wants the library name in A1 and the lowest acceptable version in D0. Parameters to the library functions are always stored in registers. See the library reference documentation for closer information exactly which registers.

This example opens a library with the name at LibName with version 31 or higher:

```
INCLUDE "exec/libraries.i"
; for exec functions
INCLUDE "exec/exec_lib.i"
; exec function index
VERSION equ 34
MOVE.L
4,a6
; exec library base
LEA
LibName,a1
; library name
MOVE.L
#VERSION,d0
; lowest usable version
JSR
_LVOOpenLibrary(a6)
; open the library in exec
```

## Libraries

The operating system provides facilities for the creation, use and access of shared libraries. The functions that let the programmer construct and access libraries are of different levels to give different possibilities. Low level function where you can change every single parameter and more high level functions that do a lot without the program more exact specification.

I'll describe the functions of the highest level that also are the most frequently used:

OpenLibrary() Gains access to a named library of a given version number.

Always open libraries with the lowest version which includes the functions you need. To open intuition library for 2.0+ (version 36) only, try something like:

```
#include <exec/types.h>
#define LIB_VERSION 36
extern struct ExecBase *SysBase;
struct IntuitionBase *IntuitionBase;
void main(void)
{
/*
 * The SysBase should be in order to use this.
 * (The startup module will do this for you.)
 */
IntuitionBase=(struct IntuitionBase *)
OpenLibrary("intuition.library", LIB_VERSION);
if(!IntuitionBase) {
printf("Couldn't open intuition version %d\n", LIB_VERSION);
exit(10);
}
/*
 * The rest using intuition.library V36+ follows here!
 */
}
```

AmigaDOS file names are not case sensitive, but Exec lists are. If the library name is specified in a different case than it exists on disk, unexpected results may occur.

# A Guide to AmigaDOS Shared Libraries

CloseLibrary()  Concludes access to a library. Whenever your program has finished using the functions of a shared library, there should be a call to CloseLibrary() for every call to OpenLibrary(). Simply like this:

```
CloseLibrary((struct Library *)IntuitionBase);
```

RemLibrary()  Calls the Expunge() function of the specified library. If the library isn't open, it will delete itself from memory. This is not typically called by user code.

```
/* Attempts to flush the named library out of memory. */
#include <exec/types.h>
#include <exec/execbase.h>
void FlushLibrary(STRPTR name)
{
struct Library *result;
Forbid();
if(result=(struct Library *)FindName(&SysBase->LibList,name))
RemLibrary(result);
Permit();
}
```

With these three functions in mind, we'll continue.

## Return Code

The return code of a shared library function call is always received in a register. (Today, I don't think there is a single function not using D0 for that purpose.)

## Glue Code

The parameter storage in registers is not that comfortable in all occasions and some compilers (in all kinds of programming languages) don't even have the ability to store parameters in (pre-decided) registers. Then, glue code is required. Glue code (also known as "stub functions" or simply "stubs") is simply a set of functions that you can call instead of the shared library functions. The stub function reads the parameters from the stack and stores them in registers and then calls the shared library function. That makes the use of the glue code functions identical to other functions. Glue code is compiled into a kind of object file, using the suffix ".lib", and is stored in LIB: (not to be mixed up with LIBS: where the shared libraries are stored). All stub functions for the standard AmigaDOS libraries are found in the "amiga.lib" file that comes with most compilers.

## C and Register Parameters

C language compilers are in general using the stack to pass parameters between functions, but to be able to use shared libraries smoothly, several compilers offer ways to force parameters in registers and automatically use the right library base and function index.

The two largest commercial C compilers on Amiga, SAS/C and Aztec C, both provide such solutions by special pragma instructions. A pragma instruction is a line starting with "#pragma", which is a compiler instruction keyword, followed by the compiler specific text. Such a pragma defines the function, which library base it needs and in which registers the parameters must be stored. By using such pragmas you don't have to call or link any glue code within your program.

The GNU C compiler, which is a freely distributable C and C++ compiler, has a very complicated way to solve this problem. It declares and uses inlined functions that use GNU's own asm() instruction to set the proper registers to the right values.

SAS/C pragmas are built up like this:

```
#pragma <kind of call> <lib base> <name> <index> <registers>
which means: #pragma
Compiler instruction keyword. <kind of call>
```

Which kind of library call should this pragma generate? There are three different ones:

"libcall" makes a standard library call

"tagcall" makes a standard library call where the last parameter points to a tag list

"syscall" makes a call to exec.library <lib base> The library base name to use. Not specified for "syscall" calls.

**Example:** "DiskfontBase" (The name of diskfont.library's library base.) <name>

Function name identifier

**Example:** "MyFunction". <index>

Function index of the library. A hexadecimal, positive number (which is turned negative by the compiler when it generates the indexed library call). **Example:** "1E" (The first library function index of all normal libraries.) <registers>

Register/parameter information in a special format, a sequence of hexadecimal numbers. Reading from the right, each digit has the following meaning:

1. Number of parameters.

2. Result code register (0-6 means register D0-D6 and 8-9, A-E means register A0-A6)

3+. The parameter registers, read from the left (!). The numbers are associated with the same registers as in paragraph 2 above.

**Example:** #pragma libcall SysBase OpenLibrary 228 0902

When using this information, a compiled result uses SysBase, the index and the parameters in registers just as we did in the assembler examples above. C language usage:

```
#include <proto/exec.h>
/* This includes the pragma too */
#define LIBNAME "foobar.library"
#define VERSION 33
OpenLibrary(LIBNAME, VERSION);
```

(Generate pragma files with the SAS/C utility "fd2pragma" which uses a function descriptor file as source of information. Read about function descriptor files further on.)

## Indexing Effects

Compilers of different programming languages often create machine language instructions that address data indexed by a 16-bit register, instead of straight 32-bit addressing, to increase execution speed and decrease the code size.

Some libraries might request or offer a "callback function", a function supplied by you in the form of a function pointer that might get called from inside the library. A call from within a library may not have that index register set properly and therefore you must set it before you can access any data that requires that register!

In SAS/C, this is simply done by defining the function like:

```
void __saveds callback(void) {
If using DICE, __saveds must be replaced with __geta4.
```

(In the SAS and Aztec compilers, it can also be done by calling geta4() first in the callback function.)

From version 36/37 some of the AmigaDOS system libraries feature hook abilities, which is a kind of callback function. They are also called from inside the library and then of course also demand loading of the index register the same way.

## Registers

Library functions should preserve the a2-a7 and d2-d7 registers. The rest must be stored in a safe place and then brought back after the library call if you want to be sure of their contents.

## Parts of an AmigaDOS Shared Library Image

If we were content with only using shared libraries, we would have enough information by now to use all kinds of library calls.

Only scratching the surface isn't enough if we want to create something by ourselves. We must instead start digging into detailed information. How is a shared library constructed? Of which parts? How do you combine those parts to make your own shared library?

First we take a look at the parts of a shared library. A shared library does not look the same when compiled/assembled as when loaded into memory and added to the system's list. That is because when the library is loaded/added it is also modified and initialized in a few ways to make the system able to use it. But let's not hesitate.

A shared library image is built up by a few different parts.

- Code preventing execution
- ROMTag structure with sub data:
  - Init table
  - Function pointer table
  - Data table
- Init routine
- Functions-Prevent Execution

The first thing the disk image contains is a piece of code that prevent users from trying to execute the library as an executable file. That piece of code should preferably return an error code to the calling environment (that most possibly is a shell).

**Example:**

```
moveq
#-1,d0
[illegible]
```

Coming up next is a ROMTag structure. ROMTags are used to link system resident modules together. The ROMTag looks like:

```
(found in exec/resident.h)
struct Resident {
UWORD rt_MatchWord;
struct Resident *rt_MatchTag;
APTR  rt_EndSkip;
UBYTE rt_Flags;
UBYTE rt_Version;
UBYTE rt_Type;
BYTE  rt_Pri;
char  *rt_Name;
char  *rt_IdString;
APTR  rt_Init;
};    rt_MatchWord -
```

Used by exec to find this structure when it is about to link us into the ROMTag list. This must contain RTC_MATCHWORD (the hexadecimal number $4AFC, which is a MC68000 "ILLEGAL" instruction).

rt_MatchTag - This must contain a pointer to this struct.
rt_EndSkip - Pointer to end of library init code.
rt_Flags - RTF_AUTOINIT informs exec that the structures rt_Init member points to an init table.
rt_Version - Library version number
rt_Type - Should contain NT_LIBRARY (found in exec/nodes.h), which informs exec about the fact that this is a shared library image.
rt_Pri - Initialization priority. 0 (zero) is perfectly ok.
rt_Name - Pointer to the zero terminated library name
rt_IdString - Standard name/version/date ID string

**Example.**

```
'mysms.library 1.0 (01.04.95)' rt_Init
```

This data points to an init table if RTF_AUTOINIT is set in structure member rt_Flags.

As you can see, this structure requires some more information stored. You must have the library name and a standard ID string stored, and the last structure member should point to a "init table".

### Init Table

The init table is a table of four long words. I try to visualize them in a structure like this:

(A struct of this kind is not found in any standard include file, this is written by me.)

```
struct InitTable {
ULONG it_LibBaseSize;
APTR  it_FuncTable;
ULONG *it_DataTable;
APTR  it_InitRoutine;
};
```

it_LibBaseSize - Size of your library base structure. In common situations there is no point in using anything else but a straight struct Library as library base. It must not be smaller than that!
it_FuncTable - This should contain a pointer to an array of function pointers.
it_DataTable - Pointer to a data table in exec/InitStruct format for initialization of the library base structure.
it_InitRoutine - Pointer to a library initialization routine or NULL.

Once again we have a structure that needs more data. The function pointer table, the data table and the init routine is left.

### Function Pointer Table

This should be a table of function pointers to the different functions in the library. They can be specified in two ways:
1) By setting the first word in the list to -1, you specify that the table is a list with 16-bit addresses relative to the start of the list. End the table with a -1 word.
2) By storing absolute 32 bit pointers to the functions and ending with a -1 long word. My examples will use the second way.

The pointers should point to the functions of the library. All libraries should still have a few standard functions used by exec and must not be left out. The first four entries are dedicated to such functions.

The list must look like:

```
- Open()
  Open library routine.
- Close()
  Close library routine.
- Expunge()
- Delete library from memory routine.
- ExtFunc()
- Reserved for future expansion.
- Func1()
- Our first function
- Func2()
- Our second function
- ...
- The rest of our functions
  :
- End of table
```

How to program such functions is discussed further on. Let's continue, we have the data table and the init routine left to look at.

### Data Table

The data table is used to initialize the library base structure when it's linked into the system list of shared libraries. The table is in the so called "exec/InitStruct" format. A data table is controlling a number of different initializing methods. In our case we just use a number of offsets (relative to the library base) and their initialization values.

```
#include exec/libraries.h
#include exec/initializers.h
#include exec/nodes.h
INITBYTE
(LN_TYPE), NT_LIBRARY
; Data type: Library.
INITLONG
LN_NAME,LibName
```

```
; the name of the library
INITBYTE
 LN_FLAGS,LIBF_SUMUSED!LIBF_CHANGED
; Set the flags that tells exec we have changed
; the library and that we allow check summing.
INITWORD
LIB_VERSION,VERSION
; init version
INITWORD
LIB_REVISION,REVISION
; init revision
INITLONG
LIB_IDSTRING,IDString
; init IDString
DC.L 0
; End of InitStruct() command table
```

If you have a larger library base than a Library struct, you might want to add more initialize entries to this table. The only thing left now to complete our ROMTag structure is the init routine.

## Init Routine

This routine gets called after the library has been allocated by exec. The library base pointer is in D0, the segment list is in A0 and SysBase in A6. This function must return the library base in D0 to be linked into the library list. If this initialization function fails, the library memory must be manually deallocated, then NULL returned in D0.

Deallocate library memory by using something like:

```
move.l
d0,a5
moveq
#0,d0
move.l
a5,a1
move.w
LIB_NEGSIZE(a5),d0
sub.l
d0,a1
add.w
LIB_POSSIZE(a5),d0
jsr
_LVOFreeMem(a6)
```

The segment list, that we receive in A0, should be stored somewhere for later access. We'll need it when the library is to be removed from memory. Note that this routine will be called only once for every time the library is being loaded into memory. That makes it perfectly ok to store the segment list simply like:

```
lea
segmbase(pc),a1
move.l
a0,(a1)
rts
segmbase: DC.L 0
```

A nice way to store this data is to extend the library base structure to hold the segment list pointer too. This was the last of the initialization part. The ROMTag structure is complete. Left in the library are the functions that it should contain.

## Functions

As mentioned before, there are four required functions that should be in all shared libraries. The rest of the functions are up to you to decide, design and make sure they receive proper data. How to code the functions and what to think of when doing so, is discussed in a chapter below.

## Libraries in the System

We know what shared libraries are and we are familiar with all data stored in the library image. We know what functions to use when we want to access libraries and we know how to call library functions. What about low level information? What is done in the system when we call OpenLibrary()? How can I check if library already is loaded and which version number that library has? How can I patch a function of an already loaded library?

## Library Opening Details

When a single OpenLibrary() is called, a lot of things happen:

1. Exec checks the already loaded libraries to see if the requested library is there. If it is, go to step 6.

2. If the library name is specified without path, it is searched for in ROM, LIBS: and then current directory, otherwise simply in the specified path. The first directory that holds a library with the name it searches for, will be the one it loads from. If the library wasn't found, return NULL. If the library was found anywhere else but in ROM, it's LoadSeg()'ed into memory. ROM libraries are already accessible.

3. Exec scans the library for the 4AFC word with a following 32-bit pointer back to it. That word is the beginning of a ROMTag structure!

4. InitResident() is called, which hopefully finds the RTF_AUTOINIT flag set in the rt_Flags member of the ROMTag structure and therefore calls MakeLibrary() which performs: Memory is allocated to fit a jump table and the library base structure. The size of the library base structure is found in the first long word of the data table. The jump table is created by a call to MakeFunctions() and is placed just before the library base in memory. The size of the allocation can be read in the library base structure (lib_NegSize + lib_PosSize).

The library base structure is initialized using the data table list and an InitStruct() call. The init routine is called with the library base pointer in D0, SysBase in A6 and the segment list pointer in A0. If NULL is returned, then the OpenLibrary() fails and returns NULL. Observe that any kind of failure in InitResident() means that the library is never added to the system.

5. AddLibrary() adds the library to the system list, making it available to programs. The checksum of the library entries will be calculated.

6. The OpenLibrary() call's version number parameter is checked against the version number of the library base (lib_Version). If the requested number is higher than the library version, OpenLibrary() fails and returns NULL.

7. The open function of the library is called. If that fails NULL is returned, otherwise the library base is returned in D0.

If the same library exists in LIBS: with one version and in current directory with a later version, OpenLibrary() will always go for the one that it finds first. In this case that is the library in LIBS:. If that library has a too low version number, OpenLibrary() fails.

As you can see, OpenLibrary() is a rather high level function. By using the other mentioned functions you can add a library to the system without going the way I describe in this article. But that wouldn't make it a standard shared library.

## Library List

Exec keeps track of all libraries that are opened. We can take part of exec's library list information by studying the linked list starting at SysBase->LibList. That pointer points to a 'struct List', whose 'struct Node' pointers point to the 'struct Library' of all libraries that are currently in memory. This sounds more difficult than it is. Take a look at this small example.

To find a certain library name in the library list, we can write:

```
struct Library *Find(char *name)
{
struct Library *lib;
Forbid();
lib = (struct Library *)FindName(&SysBase->LibList, name);
Permit();
return lib;
}
```

All libraries that are opened get a jump table created. That means that even ROM based libraries get a jump table in RAM. When using functions in any library, we always go through that jump table which consists of nothing but a number of JMP #ADDRESS. As you under-

zs.ahindu.org

stand, these jumps are supposed to jump into the library to perform whatever they are supposed to. By changing an entry in the jump table, we can make a certain library call to call our own function instead of the original! But to change an entry is more than just storing in the list (since there are checksums and things that have to be correct). The correct way to do it, is to use SetFunction(), which can make one of those JMP's jump to our own code.

To replace OpenLibrary() with our own function, we can do it like:

```
#include <exec/types.h>
#include <proto/exec.h>
int (*oldOpenLibrary)(char *, int);
void patch(void)
{
APTR oldfunc;
oldfunc = SetFunction((struct Library *)SysBase,
 -552,
 (APTR)NewOpenLibrary);
/*
 * Now all following calls to OpenLibrary() will
 * call our own function instead
 */
/*
 * To back back, we simply use SetFunction()
 * again. We really should try to check before
 * doing our restore someone else might have
 * patched the function after us, and if we
 * simply restore our original we would ruin
 * that patch.
 */
SetFunction(SysBase, -552, oldfunc);
}

int __asm NewOpenLibrary(register __a1 libName,
register __d0 version)
{
/*
 * Code for our library opener. Do remember that
 * our local registers is not initialized here, and
 * if you want to use them you can restore the
 * original value before returning from this
 * function. We don't want to crash any programs.
 * Do so?
 */
/* ... return register! */
}
```

Patching libraries are often used when creating debugging tools (such as the well known 'Mungwall' which patches AllocMem and FreeMem, 'Snoopdos' which hangs on to most of dos.library's functions and others) and for programs that enhances or somehow changes the functionality of a function system wide (such as 'Explodewindows' which patches OpenWindow() and beautifies window openings, 'RTpatch' and 'Reqchange' which catches different requester calls to bring up reqtools.library requesters instead). NOTE: SetFunction() cannot be used on nonstandard libraries like dos.library! If you want to patch dos.library, you must manually Forbid(), preserve all 6 original bytes of the jump table entry, SumLibrary() (to evaluate the new checksum) and then Permit().

## Programming Functions

Shared Libraries must be programmed by someone. Until now you've learned how to control, play around and change already existing libraries. Now, we'll check out more of what there is to know to be able to program a library. The ROMTag initializing is of course required when programming a library, but the biggest part and the part that really makes the library, is still the functions.

You're not restricted to anything when it comes to the function of the routines you want to put in a shared library. What must be thought of when creating functions for a shared library using a compiler is that there is no main() function and no startup modules, and therefore no one of the symbols declared in those modules will be declared if you don't do it yourself.

There are always four functions required that have to be in every library. They are Open(), Close(), Expunge() and Extfunc() and are called by exec when the library is to be opened, closed and removed from memory (the fourth is reserved for future use). Exec turns off task switching while executing these routines (via Forbid()), so we should make them not take too long. (When using SAS/C these functions won't be necessary to code, see the "Compiling" and "Linking" chapters.)

```
- Open()
  (library base:a6, version:d0)
```

This routine is called by exec when OpenLibrary() (or more correct InitResident()) is called. Open should return the library pointer in D0 if the open was successful. If the open fails, NULL should be returned. It might fail in cases where we allocate memory on each open, or if the library only can be open once at a time.

Example:

```
; Increase the library's open counter
addq.w
#1,LIB_OPENCNT(a6)
; Switch off delayed expunge
bclr
#LIBB_DELEXP,LIB_FLAGS(a6)
; Return library base
move.l
a6,d0

- Close()
  (library base:a6)
```

This routine is called by exec when CloseLibrary() is called. If the library is no longer open and there is a delayed expunge, then Expunge! Otherwise Close should return NULL.

Example:

```
; Decrease the library's open counter
subq.w
#1,LIB_OPENCNT(a6)
; If there is anyone still open, return
bne.s
.notlabel
; Is there a delayed expunge waiting?
btst
#LIBB_DELEXP,LIB_FLAGS(a6)
beq.s
.notlabel
; Do the expunge!
bsr
Expunge
.notlabel:
; set the return value
moveq
#0,d0

- Expunge()
  (library base:a6)
```

This routine is called by exec when RemLibrary() is called, or from Close when there was a delayed expunge. If the library is no longer open then Expunge should Remove() itself from the library list, FreeMem() the InitResident()'s allocation and return the segment list (which was given to the Init routine). Otherwise Expunge should set the delayed expunge flag and return NULL.

Because Expunge might be called from the memory allocator, it may NEVER Wait() or otherwise take long time to complete.

Example:

```
; Is the library still open?
tst.w
LIB_OPENCNT(a6)
beq
.close
; It is still open. Set the delayed expunge flag
; and return zero
bset
#LIBB_DELEXP,LIB_FLAGS(a6)
moveq
#0,d0
rts
.close:
movem.l d2/a5-a6,-(sp)
```

Dcomputers.cz

```
; save some registers
movem.l
d1,a7
; Store the segment list in a6
lea
segment(pc),a6
move.l
(a6),d2
move.l
4,a6
; get SysBase
; UnLink from library list
move.l
a5,a1
jsr
_LVORemove(a6)
; This removes our
; node from the list
; Free our memory
moveq
#0,d0
move.l
a5,a1
move.w
LIB_NEGSIZE(a5),d0 ; get table size
sub.l
d0,a1
add.w
LIB_POSSIZE(a5),d0 ; the size of the rest
; of the library.
jsr
_LVOFreeMem(a6)
; Return the segment list
move.l
d2,d0
movem.l
(sp)+,d2/a5/a6
; Get back the registers
.LibExtFunc()
; ...[illegible]
```

This routine is reserved for future use and should return 0 in register D0.

Example:

```
moveq #0,d0
```

## RTS Function Descriptor File

To easily use the SAS/C options for creating a shared library, a standard AmigaDOS function descriptor file is required. It describes the functions in a library like:

```
##base _OurBase
##bias 30
OurFunction1(a0,a1/d0)
OurFunction2(d0,d1)
##end
##private
##public
```

The library base identifies ##bias - Index base position. The first function specified will use this index, which should be positive (turned negative later by the compiler) and in all normal cases starts at the first free jump table entry: 30

functionname(a0)(registers) Describes in which registers the function received its parameters in. The registers should be written without any spaces between them as in the example above.

##end - end of function descriptor file Glue Code: Glue code is written to be called with the parameters on the stack instead of the registers as it should. The glue function should pick parameters from the stack and assign to the proper registers.

Example:

MOVE.L

```
A6,-(sp) ; Store register A6 on stack
MOVE.L
A5,-(sp) ; Store register A5 on stack
MOVE.L
12(sp),A0 ; Get first argument from stack
MOVE.L
16(sp),A1 ; Get second argument from stack
MOVE.L
4,A6 ; Get SysBase in A6.
jsr
_LVOOpenLibrary(A6) ; call OpenLibrary()
; Now d0 contains the result of the
; library call
MOVE.L
(sp)+,A5 ; Restore A5 from stack
MOVE.L
(sp)+,A6 ; Restore A6 from stack
rts
```

Things to think of when compiling library code:

Always make the function called from another process (the outside) a "__saveds" function as the index register has to be properly initialized before continuing. __saveds should be replaced with __geta4 when using Dice and an initial GetA4() call when using Manx.

Whether to use global symbols unique or shared by every task. SAS/C features easy changing between these two, but other compilers might have trouble dealing unique global variables for each library open.

Options when compiling a library may include some of the following. (These are the SAS/C options, but all compilers of today offer similar functionalities.)

LIBCODE Forces all index addressing to use the library base pointer (a6) instead of the standard a4.

NOSTANDARDIO Do not use any of the C standard io functions such as printf() or fopen() etc., since they rely on global symbols declared and initialized in the startup module.

OPTIMIZE Optimize the output code. Linking: Linking a library often causes many problems, at least it has done so for me. You must remember that no compiler startup symbols will exist unless you declare them! Things like stack expansion can't be made to work, and routines like fopen() and others are using startup module symbols (which can be declared by us though).

With the symbols in mind, we continue! All the talk about the library initializer structure is no problem of a SAS/C programmer's mind. By including the following flags in your 'slink' line, all such problems are solved:

### LIBPREFIX <prefix>

Default is "_" (underscore). This is the prefix added to the functions specified in the function descriptor file to match the symbols of the object file(s).

### LIBFD <function desc file>

Tells where the function descriptor file is.

### FROM lib:libent.o lib:libinit.o

Two nice object files holding code that we would have to code by ourselves otherwise. If you are using global variables in your code, "libinit.o" will make all current library openers of the library access the same, shared, variable. By using "libinitr.o" all globals will be copied at the library open, thus each open library has its own global variables.

### LIBID

Sets the IdString of the library

### LIBVERSION <number>

Sets the version number of the library

### LIBREVISION <number>

Sets the revision number of the library Debugging

Using SAS/C, shared libraries can be run time debugged (including variable checking, break pointing and so on), just like any other program using the "step into reslib" option in 'cpr'. Break any library function by writing "b mylibrary.library/foobar" (where foobar is the name of the

function we want up to stop to when we enter the for command prompt after. When creating debug code, remember to debug the library that exists in the same directory as the code does, or specify the compiler flag SOURCEIS= and the name of your sourcefile.

### Hints

I have been programming and developing shared libraries for some time by now, and there are a few things to pay certain attention to when dealing with this stuff.

### Flush before retry

Libraries don't go away simply because you close them, you know that. If you run your library once, close it and recompile it with a few changes, there will still be the older version remaining in memory that will be opened. When debugging libraries, always make sure that your library isn't already in memory before debugging a new version!

I made a small program that reads the open counter and then RemLibrary()'s the named library. It's not at all a nice thing to do, but there really is a problem when you open your library and something crashes before you have had the chance to close it. There is no "nice" way of removing such a library from memory!

### Globals

By using the SAS/C object files libinit.o or libinitr.o you can make your global variables to be shared by all processes or unique for each OpenLibrary() call. If you want to mix the two versions or create something different, I advise you to code the library initial code by yourself.

### Stack usage

When your library is called and runs, it uses the same stack as the caller. If the caller has a very small stack, so do you. Built-in stack check routines are not available since they need irreplaceable symbols. For advanced users, allocating and using an own stack while the library is running could be the only and best way to solve a problem like this.

### Symbols

I've written it earlier and I do it again: high level language functions often uses symbols initialized and declared in the startup modules. Declare them by yourself if possible or avoid using such functions!

### Register preservation

I think it's a good habit to always preserve all registers (except for D0 that holds the return code) when your library routines are called. Remember that your library code index register is uninitialized when called from the library opener.

## Appendix A.

## VERSION NUMBERS AND SHARED LIBRARIES

Commodore has introduced a general standard for shared library version numbering. The libversion number is the true version of the library version. The librevision number is expected to be a counter from 0 and upwards, without any kind of preceding zero. This makes the first library version 1.0 and such as 1.9 is followed by 1.10, 1.11 and so on all the way to the maximum, of the same version, 1.65535.

Failing in the version number check while library opening leaves the library in memory. For example, when you want to open "mysown.library", it's loaded into memory. If the version number check fails and you get a NULL in return, "mysown.library" will still remain loaded.

There are utilities which automatically updates a source file with the version number, revision number and the ID string on every invoke. bumprev is one.

## B. FURTHER READING

Amiga ROM Kernel Reference Manual: Libraries, 3rd edition

Amiga ROM Kernel Reference Manual: Includes & Autodocs, 3rd edition

## C. LIBRARY SOURCE EXAMPLES

```
makefile.a
----
# This makefile uses the standard way of making a
# shared library with SAS/C. Using the already
# created object files [illegible]
CC
# [illegible]
INCLUDE = mysown.h
[illegible] = [illegible]
[illegible]
# [illegible]
LIBNAME = mysown.library
[illegible]
# [illegible]
[illegible]
VERSION
$(LIBNAME): [illegible]
[illegible]
[illegible] mysown.fd
[illegible]
[illegible] lib:libinit.o [illegible]
[illegible]
SD SC
[illegible] "mysown.library 2.1 [illegible]"
libversion 2 [illegible]
[illegible]
copy $(LIBNAME) [illegible] # Copy library to [illegible]
[illegible]
[illegible]
makefile.b
-----
# This makefile compiles everything and [illegible]
# pre compiled files.
# Easy changed to the DICE, Aztec or other
# compilers.
CC
# [illegible]
INCLUDE = mysown.h
SOURCE = mysown.c
ASOURCE = mysownlib.a
[illegible]
  mysown.o mysownlib.o
LIBRARY = mysown.library
[illegible]
# [illegible]
DATA [illegible]
[illegible]
ASM
# [illegible]
[illegible]
$(LIBRARY): $(OBJS)
[illegible] $(LIBRARY) [illegible]
copy $(LIBRARY) [illegible]
[illegible]: $(SOURCE) $(INCLUDE)
$(CC) $(CFLAGS) $*.c
[illegible]: $(ASOURCE)
$(ASM) $(AFLAGS) $*.a
[illegible]
[illegible]
*************************************************
*
* mysownlib.a assembler source code
*
*************************************************
* Author: Daniel Stenberg
*************************************************
SECTION
code
[illegible]
INCLUDE "exec/types.i"
INCLUDE "exec/initializers.i"
INCLUDE "exec/libraries.i"
INCLUDE "exec/lists.i"
INCLUDE "exec/alerts.i"
INCLUDE "exec/resident.i"
INCLUDE "libraries/dos.i"
[illegible]
XDEF InitTable
XDEF Open
XDEF Close
XDEF Expunge
```

# A Guide to AmigaDOS Shared Libraries

```
[illegible]
```

Dcomputers.cz

```
[illegible]
; remove
notdeleted: [illegible]
move.l
[illegible]
move.l
a5,a3
; Store our seglist list in d2
lea
seglist(pc),a0
move.l
(a0),d2
move.l
4,a6
; get SysBase
; Unlink from library list
move.l
a5,a1
jsr
_LVORemove(a6) ; This removes our node
; from the list
; free our memory
moveq
#0,d0
move.l
a5,a1
move.w
LIB_NEGSIZE(a5),d0
sub.l
d0,a1
add.w
LIB_POSSIZE(a5),d0
jsr
_LVOFreeMem(a6) ; This frees the memory
; we occupied
; Return the segment list
move.l
d2,d0
movem.l
(sp)+,[illegible] ; Get back the registers
rts

Extfunc:
; should return zero
moveq
#0,d0
rts

; EndCode is a marker that show the end of our
; code.
EndCode:
end
mysum.h
=======
/*********************************************
*
* mysum.h library source file
*
*********************************************
* Author: Daniel Stenberg
*********************************************/
/* library function prototypes */
int Min(int, int); /* return minimum value */
int Abs(int);
/* return absolute value */
mysum.c
=======
/*********************************************
*
* mysum.library functions source code
*
*********************************************
* Author: Daniel Stenberg
*********************************************/
/*
* Use the __asm [illegible] and the __D0 registers to
* force parameters into certain registers in SAS/C.
*
* Use __d0 registers to force register parameters using [illegible]
*/
int __asm Min(register __d0 int a,
register __d1 int b)
{
int c = a < b ? a : b;
return (c);
}
int __asm Abs(register __d0 int a)
{
int c = a < 0 ? -a : a;
return (c);
}
mysum.fd
========
##base _MyBase
##bias 30
Min(a,b)(D0/D1)
Abs(a)(D0)
##end
mysum_pragmas.h
===============
/* SAS/C pragmas */
#pragma libcall MyBase Min 1E 1002 /* d0 and d1 */
#pragma libcall MyBase Abs 24 001 /* only d0 */
uselibrary.c
============
#include "mysum_pragmas.h" /* if using SAS/C [illegible] */
#include "mysum.h"
struct Library *MyBase NULL;
void main(void)
{
int min, abs;
MyBase=OpenLibrary("mysum.library", 0);
if(MyBase) {
min = Min( 4, 5 ); /* library Min() function */
abs = Abs( -12 ); /* library Abs() function */
CloseLibrary( MyBase );
} else
printf("Couldn't open mysum.library\n");
}
```

Complete source code & listings can be found on the AC's TECH disk.

*Please write to:*
*Daniel Stenberg*
*c/o AC's TECH*
*P.O. Box 2140*
*Fall River, MA 02722*

# AC's Back Issue Index

# Amazing Computing

Vol 7, No 11, November 1992
Highlights Include:
"Amiga 4000," Commodore [illegible] in Amiga computing with expanded graphic resolutions, [illegible] CPU, and more.
"Progressive 040/3000," a review by Rick Manasa.
"Ramp Magic," [illegible] for making use of your palette.
"Beginning C," [illegible] covers some of the basics of the C language.

Vol 7, No 12, December 1992
Highlights Include:
"Polishing Basic Programs," [illegible] BASIC programming experts.
"Memory," [illegible]
"Structured Thinking & True BASIC," [illegible] True BASIC fully supports [illegible] structure.
Also complete reviews of Voyager 1.1, [illegible] 2.0, and OpalVision.

Vol 8, No 1, January 1993
Highlights Include:
"Creating a Storyboard in Final Copy," see how to layout your [illegible] in Final Copy by R. Shamms Mortier.
"A Look at PAGE Libraries," Shamms Mortier [illegible]
"Using Laser Disk Players with the Amiga," [illegible] video images. [illegible]
Plus: A complete review of the new A1200 & coverage of Comdex Fall 92 & the FES London.

Vol 8, No 2, February 1993
Highlights Include:
"Extending the AMOS Seal," [illegible] the AMOS [illegible] function.
"Business Cards," [illegible] business cards.
"AD1012," a review by Rick Manasa.
AND: A special sneak preview of the One-Stop Music Shop from Blue Ribbon & complete coverage of the WOC & Toronto!

Vol 8, No 3, March 1993
Highlights Include:
"Babylon 5," the Amiga [illegible] TV show are revealed by Paul Tolliver.
"AmigaVision Projects," by William Murphy.
"Art Expression," review by Merrill Callaway.
PLUS: Creative Business Forms & CES Winter '93.

Vol 8, No 4, April 1993
Highlights Include:
"Triple Play Plus & SyncPro," [illegible] by Rick Manasa.
"CanDo," a review of the [illegible] INOVAtronics by Rob Hays.
ALSO: Super Video Tips for April, [illegible], and [illegible]

Vol 8, No 5, May 1993
Highlights Include:
"Directory Opus," review of the latest [illegible] Directory Opus and a start up tutorial by Merrill Callaway.
"Media Madness," [illegible] by [illegible] & David Miller.
"SuperJAM! 1.1," a review of the latest release of SuperJAM! by Rick Manasa.
"ImageFX," review by R. Shamms Mortier.
ALSO: Super Video Tips for May—The New Graphics Market.

Vol 8, No 6, June 1993
Highlights Include:
"AMOS Turns Professional," [illegible] development system, by [illegible]
"Searching Medical Literature," using the Amiga [illegible] resources of medical [illegible], by Dr. Michael Tobin.
ALSO: Newsletter Design, ARexx Programming, [illegible]

Vol 8, No 7, July 1993
Highlights Include:
"TypeSMITH 1.0," review of Soft Logik's new font editor, by Merrill Callaway.
"DeluxePaint [illegible]," review of the latest version of the paint program for the AGA chipset, by R. Shamms Mortier.
"Structured Drawing," [illegible] by Dan Weiss.
"[illegible] AGA," review of the [illegible] package for the AGA machines, by R. Shamms Mortier.
ALSO: Super VideoTips, ARexx, and New Products.

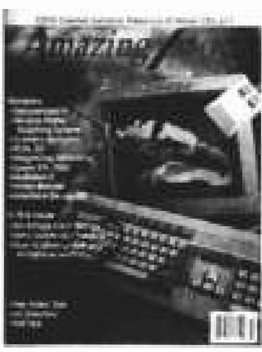
Vol 8, No 8, August 1993
Highlights Include:
"Amiga Vision Professional," review: Commodore's upgraded authoring system by Douglas L. Nakakihara.
"Art Department Professional 2.3," review of the latest version of [illegible] AGA [illegible] by Merrill Callaway.
"Professional Page 4.0," [illegible] by Rick Manasa.
"[illegible] Reality [illegible]," [illegible] by [illegible]
"T-Rexx Professional," a review of the latest version of T-Rexx [illegible] ARexx by Merrill Callaway.
ALSO: AC Phone Book: A directory of Amiga Developers.

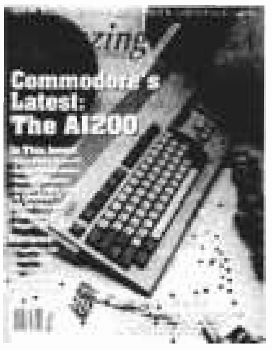
Vol 8, No 9, September 1993
Highlights Include:
"Adventures with the [illegible]," [illegible] by R. Shamms Mortier.
"CanDo," [illegible] by Randy Finch.
"Imagine [illegible]," [illegible] by R. Shamms Mortier.
"Coming Attractions," [illegible] in Amiga [illegible]
ALSO: WOCA—[illegible] & Summer CES.

Vol 8, No 10, October 1993
Highlights Include:
"Making Waves," [illegible] by R. Shamms Mortier.
"Clouds in Motion," [illegible] animation by R. Shamms Mortier.
"Media Madness," Discover what [illegible] by Rick Manasa.
"[illegible] Professional 2.0," review by Rick Manasa.
"Terminal [illegible]," A review of [illegible]
ALSO: Commodore's new [illegible]

Vol 8, No 11, November 1993
Highlights Include:
"CanDo," this installment covers creating a custom object by combining several standard CanDo objects, by Randy Finch.
"Brilliance," A complete review of the [illegible] paint and animation program from Digital Creations by Frank McMahon.
"[illegible]," The introduction of the new [illegible] column by Bill [illegible]
"Get Graphics Digital Image [illegible]," the introduction of AC's new graphics column by William Frawley.
"[illegible]," [illegible] by [illegible]
ALSO: WOCA Pasadena: Commodore introduces CD32. Plus, the [illegible] LightRave, a Video Toaster [illegible]

Vol 8, No 12, December 1993
Highlights Include:
"CanDo Tutorial," [illegible] by Randy Finch.
"LightRave Review," A review of the unique Toaster [illegible] by [illegible]
"[illegible]," [illegible] for the Amiga by [illegible]
"Get Graphics Digital Image FX," [illegible] of AC's new graphics column by William Frawley.
"Video Toaster 4000 Review," A review of the latest Video Toaster by Shamms Mortier.
ALSO: 1993 Reader's Choice Awards!



# AC's TECH

# AC's TECH, Vol. 2 No. 4
Highlights Include:
"In Search of the Lost Windows," by Phil Burke
"No Messing Around," [illegible] with this [illegible], by Jeff [illegible]
"The Joy of Sets," by Jim Olinger
"QuarterbackStat" review by Merrill Callaway

# AC's TECH, Vol. 3 No. 1
Highlights Include:
"Amiga Computing's C[illegible]," A review of the great new C compiler by [illegible]
"Programming the Amiga in Assembly Language Part 5," by William Nee
"Make Your Own 3D Vegetation," Laura Morrison shows how to use fractal functions to create 3D trees and plants
PLUS! The HotLinks Developer's Toolkit ON-DISK!

# AC's TECH, Vol 3, No. 2
Highlights Include:
"Oloi," An arcade game programmed in AMOS BASIC, by [illegible] Dikeman
"Programming the Amiga in Assembly Language Part 6," by William Nee
"Wrapped Up with True BASIC," Text and Graphics wrapping routines in True BASIC, by Dr. Roy M. Nuzzo
"A Rexx Disk Cataloger," An AmigaDOS manipulator that produces a text file containing information about the floppy disks you want cataloged, by T. Darrell Westbrook
AND LOTS MORE ON DISK!

# AC's TECH, Vol. 3 No. 3
Highlights Include:
"Rexx Rainbow Library," A review by Merrill Callaway
"Programming the Amiga in Assembly," by William Nee
"All You Ever Wanted to Know About Morphing," An in depth look at morphing for Imagine by Bruno Costa and Lucia Darsa
"Custom 3D Graphics Package Part I," Designing a custom 3D graphics package by Laura Morrison
"Build a Second Joystick Port," A simple hardware project for an additional joystick port by [illegible]
AND LOTS MORE ON DISK!

# AC's TECH, Vol. 3, No. 4
Highlights Include:
"Custom 3D Graphics Package Part II," Put the finishing touches on your own graphics package by Laura Morrison
"TrueBASIC Input Mask," An interesting TrueBASIC utility by T. Darrell Westbrook
"Time Efficient Animations," [illegible] utility by Robert Galka
"F-BASIC 5.0," A review of the latest version of F-BASIC by Jeff Stein
PLUS: CD32 Development Info!

# 1-800-345-3360

**BACK ISSUE SPECIALS! CALL FOR DETAILS**

**Complete selection of Amazing Computing and AC's TECH AVAILABLE!**

**WHAT HAVE YOU BEEN MISSING?** Have you missed information on how to add ports to your Amiga for under $20, how to work around *Deluxe Paint's* lack of HAM support, how to deal with service bureaus, or how to put your Super 8 films on video tape, along with Amiga graphics? Do you know the differences among the big three DTP programs for the Amiga? Does the ARexx interface still puzzle you? Do you know when it's better to you use the CLI? Would you like to know how to go about publishing a newsletter? Do you take full advantage of your RAM disk? Have you yet to install an IBM mouse to work with your bridgeboard? Do you know there's an alternative to high cost word processors? Do you still struggle through your directories?

Or if you're a programmer or technical type, do you understand how to add 512K RAM to your IMB A500 for a cost of only $40? Or how to program the Amiga's GEL in C? Would you like the instructions for building your own variable rapid fire joystick or a 256-grayscale SCSI interface for your Amiga? Do you use easy routines for performing floppy access without the aid of the operating system? How much do you really understand about ray tracing?

**The answers to these questions and others can be found in *AMAZING COMPUTING* and *AC's TECH*.**

DJComputers.cz

# SUBSCRIBE!

**YES!** *The "Amazing" AC publications give me 3 GREAT reasons to save!*

*Please begin the subscription(s) indicated below immediately!*

Name ____________________ ____________________

Address ____________________

City ____________________ State ______ ZIP ______

Charge my Visa MC # ____________________

Expiration Date ______ Signature ____________________

Please circle to indicate this is a **New Subscription** or a **Renewal**

MasterCard · VISA · DISCOVER

Call now and use your Visa, Master Card, or Discover or fill out and send in this order form

| | | |
|---|---|---|
| 1 year of AC | 12 big issues of Amazing Computing! Save over 43% off the cover price! | **US $27.00** ☐<br>**Canada/Mexico $34.00** ☐<br>**Foreign Surface $44.00** ☐ |
| 1-year SuperSub | AC + AC's GUIDE - 14 issues total! Save more than 45% off the cover prices! | **US $37.00** ☐<br>**Canada/Mexico $54.00** ☐<br>**Foreign Surface $64.00** ☐ |
| 1 year of AC's TECH | 4 big issues of the FIRST Amiga technical magazine with Disk! | **US $43.95** ☐<br>**Canada/Mexico $47.95** ☐<br>**Foreign Surface $51.95** ☐ |

Please call for all other Canada/Mexico/foreign surface & Air Mail rates

Check or money order payments must be in US funds drawn on a US bank; subject to applicable sales tax.

---

# ORDER!

NAME ____________________

ADDRESS ____________________ ____________________

CITY ____________ ____________ STATE ________ ZIP ______

CHARGE MY: VISA M/C # ____________________

EXPIRATION DATE ____________ SIGNATURE ____________

MasterCard · VISA · DISCOVER

**Amazing Computing Back Issues:** $5.00 each US, $6.00 each Canada and Mexico
$7.00 each Foreign Surface. Please list issue(s) ______

**Amazing Computing Back Issue Volumes:**
Volume 1-$15.00* Volume 2, 3, 4, 5, 6, 7, or 8 - $20.00* each or any 12 issues for $20.00*
* All prices now include shipping & handling. * Foreign surface: $25. Air mail rates available

**AC's TECH AMIGA** Single issues just $14.95! V1.1 (Premiere), V1.2, V1.3, V1.4, V2.1, V2.2, V2.3, V2.4, V3.1, V3.2, V3.3, V3.4
Volume One, Two, or Three (complete) or any four issues— $40.00

**Freely Distributable Software – Subscriber Special (yes, even the new ones!)**

| | |
|---|---|
| 1 to 9 disks | $6.00 each |
| 10 to 49 disks | $5.00 each |
| 50 to 99 disks | $4.00 each |
| 100 or more disks | $3.00 each |

**$7.00 each for non subscribers** **(three disk minimum on all foreign orders)**

Amazing on Disks:

AC#1 .. Source & Listings V3.8 & V3.9
AC#2 .. Source & Listings V3.10 & V4.1
AC#3 .. Source & Listings V4.5 & V4.6
AC#4 .. Source & Listings V4.7 & V4.8
AC#5 .. Source & Listings V4.9
AC#6 .. Source & Listings V4.10 & V4.11
AC#7 .. Source & Listings V4.12 & V5.1
AC#8 .. Source & Listings V5.2 & V5.3
AC#9 .. Source & Listings V5.4 & V5.5
AC#10 ...Source & Listings V5.6 & 5.7
AC#11 ...Source & Listings V5.8, 5.9 & 5.10
AC#12 ...Source & Listings V5.11, 5.12 & 6.1
AC#13 ...Source & Listings V6.2 & 6.3
AC#14 ...Source & Listings V6.4 & 6.5
AC#15 ...Source & Listings V6.6, 6.7, 6.8 & 6.9

Bock issues $ ______
AC's TECH $ ______
PDS Disks $ ______
Total: $ ______

Please list your Freely Redistributable Software selections below:

*AC Disks* ____________________
*(numbers 1 through 15)*

*AMICUS* ____________________
*(numbers 1 through 26)*

*Fred Fish Disks* ____________________
*(numbers 1 through 910)*

***Complete Today, or telephone 1-800-345-3360 now!***

***You may FAX your order to 1-508-675-6002***

Please allow 4 to 6 weeks for delivery of subscriptions in US.

*(Domestic and Foreign air mail rates available on request)*

Check or money order payments must be in US funds drawn on a US bank; subject to applicable sales tax.

# High Resolution Output

## from your AMIGA™ DTP & Graphic Documents

DJComputers.cz

You've created the perfect piece, now you're looking for a good service bureau for output. You want quality, but it must be economical. Finally, and most important...you have to find a service bureau that recognizes your AMIGA file formats. Your search is over. Give us a call!

We'll imageset your AMIGA graphic files to RC Laser Paper or Film at 2400 dpi (up to 154 lpi) at a extremely competitive cost. Also available at competitive cost are quality Dupont ChromaCheck™ color proofs of your color separations/films. We provide a variety of pre-press services for the desktop publisher.

**Who are we?** We are a division of PiM Publications, the publisher of *Amazing Computing for the Commodore AMIGA*. We have a staff that *really* knows the AMIGA as well as the rigid mechanical requirements of printers/publishers. We're a perfect choice for AMIGA DTP imagesetting/pre-press services.

*We support nearly every AMIGA graphic & DTP format as well as most Macintosh™ graphic/DTP formats.*

*For specific format information, please call.*

*For more information call 1-800-345-3360*

*Just ask for the service bureau representative.*

# *Special Offer for AC Readers!*

## AMOS (US), AMOS Compiler, and AMOS 3D

all three for only **$99.99***

# Bring your Amiga to *Life!*

**AMOS - The Creator** is like nothing you've ever seen before on the Amiga. If you want to harness the hidden power of your Amiga, then AMOS is for you!

AMOS Basic is a sophisticated development language with more than 500 different commands to produce the results you want with the minimum of effort. This special version of AMOS has been created to perfectly meet the needs of American Amiga owners. It includes clearer and brighter graphics than ever before, and a specially adapted screen size (NTSC).

"Whether you are a budding Amiga programmer who wants to create fancy graphics without weeks of typing, or a seasoned veteran who wants to build a graphic user interface with the minimum of fuss and link with C routines, AMOS is ideal for you." *Amazing Computing, June 1992*

HERE ARE JUST SOME OF THE ▼ THINGS YOU CAN DO ▼

- Define and animate hardware and software sprites (bobs) with lightning speed
- Display up to eight screens on your TV at once - each with its own color palette and resolution (including HAM, interlace, half brite and dual playfield modes)
- Scroll a screen with ease. Create multi-level parallax scrolling by overlapping different screens - perfect for scrolling shoot em ups
- Use the unique AMOS Animation Language to create complex animation sequences for sprites, bobs or screens which work on interrupt
- Play Soundtracker, Sonix or GMC (Games Music Creator) tunes or IFF samples on interrupt to bring your programs vividly to life
- Use commands like RAINBOW and COPPER MOVE to create fabulous color bars like the very best demos
- Transfer STOS programs to your Amiga and quickly get them working like the original
- Use AMOS on any Amiga from an A500 with a single drive to the very latest model with hard disk.

## WHAT YOU GET!

**AMOS (US)**—AMOS BASIC, sprite editor, Magic Forest and Amosteroids arcade games, Castle AMOS graphic adventure, Number Leap educational game, 400-page manual with more than 40 example programs on disk, sample tunes, sprite files, and registration card.

**AMOS Compiler**—AMOS Compiler, AMOS language updater, AMOS Assembler, eight demonstration programs which show off the power of the compiler, and a comprehensive, easy-to-use manual to develop lightning fast software.

**AMOS 3D**—Object Modeler, 30 new AMOS commands, and more. AMOS 3D allows you to create 3D animations as fast as 16 to 25 frames per second. You can display up to 20 objects at once, mix 3D with other AMOS features such as sprites, bobs, plus backgrounds, and more.

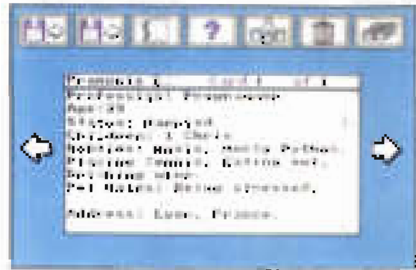
### *Limited Time Offer for AC readers only!*

Get all three AMOS packages at one great price. Order today by sending your name, address (physical address please—all orders will be shipped by UPS), and $99.95 (*plus $10.00 for shipping and handling) to: AMOS Special, PiM Publications, Inc., P.O. Box 2140, Fall River, MA 02722-2140 or use your VISA, MasterCard, or Discover and fax 1-508-675-6002 or call toll free in the US or Canada:

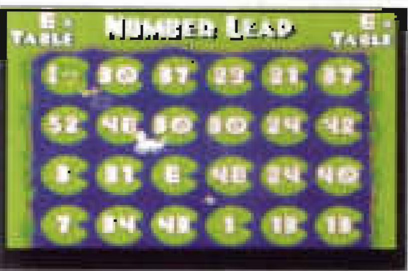
**1-800-345-3360**

Please allow 4 to 6 weeks for delivery. AMOS written by François Lionet. ©1992 Mandarin/Jawx. Europress Software, UK

Use the sophisticated editor to design your creations



Create serious software like Databases





Produce educational programs with ease

Play Magic Forest and see just what AMOS can do!

Design sprites using the powerful Sprite Editor

Create breathtaking graphical effects as never before