ARTv1i3

Osma Ahvenlampi

COLLABORATORS			
	TITLE : ARTv1i3		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Osma Ahvenlampi	July 20, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

ARTv1i3

1.1 No title

Amiga Report Technical Journal

Amiga Report Technical Journal Volume 1 Issue 3

In this issue

Departments
* Editorial
* Pointers -- Hot spots, news, and files
* Reader~mail
* C~Programming, a new column by Ken Howe

Features
* First part of an Amiga~E~course, with Sebastian Rittau as the instructor
* Introduction to Flex~and~Bison, by Joe C. Solinsky
* Robert Reiswig continues the Installer~series
* Josef Faulkner's ARexx~tutorial
* The ARTech~survey~#1~results
* Chris Aldi's BOOPSI~guide continues

Producing Editor, Osma Ahvenlampi, Osma.Ahvenlampi@hut.fi

Supervising Editor, Jason Compton, jcompton@xnet.com, FS Publications.

Contributors, staff, and contact addresses

1.2 Editorial

Never depend on computers. This issue is late for a combination of reasons, mostly because of a HD crash that forced me to start over with an already tight schedule. Never depend on your site adminstration's backup schedules either, do your backups independently. Failures always happen at the worst possible moment.

To make up for the long delay, in this issue we a have a lot of interesting stuff. An article revealing some of the secrets behind the often-heard-of but little known development tools flex~and~bison (or lex and yacc), a new~column~about~C~programming, the first part of an AmigaE~course, an ARexx~tutorial, and more Installer tips and BOOPSI~magic.

While Amiga Technologies still keeps us waiting for anything tangible, be it in the way of hardware or even concrete information about the future, the independent developer scene is still going strong. Every once in a while I'm about to lose the hope of Amiga surviving, but usually it doesn't take more than a quick look on Aminet to see that the platform is very much alive. A big thanks and good luck to all the people who are keeping it that way. Many exciting new programs have appeared since the last issue, some of them mentioned in the pointers section. I'm sorry I can't list them all, but that alone would be almost a full-time job ;)

Yours,

Osma Ahvenlampi, editor of Amiga Report Technical Journal.

1.3 The latest funky interests

```
  /* Pointers.c */

#include "pointerdefs.h"

ULONG Main(int Attraction) {
    Object *Pointers = NULL;

    Pointers = NewObject(PointerClass, NULL,
            POINTER_FTP,    NewObject( FTPPointerClass, NULL,
                    FTP_Site,       AMINET,
                    FTP_Filename,   "dev/gui/ClassActDemo.lha",
                    FTP_ShortDesc,  "ClassAct, Font adaptive BOOPSI GUI toolkit",
                    FTP_FullDesc,   "dev/gui/ClassActDemo.readme",
                    FTP_Flags,      FTP_COOL | FTP_BOOPSI | FTP_TOOL |  ↩
                       FTP_DEVELOPMENT,
                    TAG_DONE),

            POINTER_FTP,    NewObject( FTPPointerClass, NULL,
                    FTP_Site,       AMINET,
                    FTP_Filename,   "comm/tcp/FTPMount.lha",
                    FTP_ShortDesc,  "Mounts FTP sites as part of a filesystem",
                    FTP_FullDesc,   "comm/tcp/FTPMount.readme",
                    FTP_Flags,      FTP_NEAT | FTP_NETWORK | FTP_TOOL,
                    TAG_DONE),

            POINTER_FTP,    NewObject( FTPPointerClass, NULL,
                    FTP_Site,       AMINET,
                    FTP_Filename,   "gfx/edit/Iconian2_94.lha",
                    FTP_ShortDesc,  "OS3.0 icon editor, NewIcon support.",
                    FTP_FullDesc,   "gfx/edit/Iconian2_94.readme",
                    FTP_Flags,      FTP_ICONS | FTP_WORKBENCH | FTP_UTILITY,
                    TAG_DONE),

            POINTER_FTP,    NewObject( FTPPointerClass, NULL,
                    FTP_Site,       AMINET,
                    FTP_Filename,   "text/edit/QuillDemo.lha",
                    FTP_ShortDesc,  "Demo of Digital Quill, NEW text editor",
                    FTP_FullDesc,   "text/edit/QuillDemo.readme",
                    FTP_Flags,      FTP_DEMO | FTP_EDITOR | FTP_COMMERCIAL,
                    TAG_DONE),

            POINTER_FTP,    NewObject( FTPPointerClass, NULL,
                    FTP_Site,       AMINET,
                    FTP_Filename,   "text/print/HWGPOSTbeta7.lha",
                    FTP_ShortDesc,  "PostScript Library with many Level 2 features ↩
                       ",
                    FTP_FullDesc,   "text/print/HWGPOSTbeta7.readme",
                    FTP_Flags,      FTP_LIBRARY | FTP_GRAPHICS | FTP_PRINTER,
                    TAG_DONE),

            POINTER_FTP,    NewObject( FTPPointerClass, NULL,
                    FTP_Site,       "max.physics.sunysb.edu",
                    FTP_Filename, "pub/amosaic/AMosaic20Prerelease3_AmiTCP.lha",
                    FTP_ShortDesc,  "AMosaic 2.0 pre 3 for AmiTCP",
```

```
                    FTP_FullDesc,   "pub/amosaic/README",
                    FTP_Flags,      FTP_WEB | FTP_NETWORK | FTP_READER,
                    TAG_DONE),

            TAG_DONE);

    if(Pointers)
    {
        DisposeObject(Pointers);
        return ((FUNKY | COOL | THRIVING));
    }
    else return(NULL); }  /*
*   If you would like to see your pointer entered into next month's Main( ↩
    Attraction),
*   EMail your struct definition to artech@warped.co.
*/
```

1.4 The mail room

```
    From: "tinic urou" tinic@tinic.mayn.sub.de

  I have just read you Issue 2 of the ARTech and I can just say: FINE!

  Although there are not much articles (The copyright notice looks
  bigger than the articles 8)), its very interesting. Where can I get
  the "pointerdefs.h" ?? 8))

  It could be interesting if you could include a reference guide for
  Amiga-Programmers. I always had the problem to get examples sources
  and documentations for special hardware and software.

  Examples:
    * Where can I get sources of an example implementation of a
      Halftone-Dithering? Who comes to the idea, looking at the
      Sourcecode of "SpecialHost for AmigaTeX" which includes nearly all
      dither-routines?
    * How to get a documentation for SCSI programming? Did you really
      know that under "http://www.abekrd.co.uk/SCSI2/" you find a
      complete reference?
    * Where to get a documentation for a scanner?
    * Where can I find good examples of programming Msg-Ports? Whats the
      filename in this archive?
    * Where to get documenation for a XXX-algorithm and where can I find
      example sources?

  There are many other questions. A list which could be completed every
  issue would be nice.

  An article how to make good AmigaGuide dcumentations and online helps
  and when to use them, would also be nice, i hate bad docs!

  -

  [Good suggestions. Would someone like to volunteer to maintain a such a
  reference?]
```

From: "Victor Ramamoorthy S3 Inc (408)980-5401 x3279" vrm@s3.com

I have read both the issues of ARTech and found it to be interesting. Can you please make the articles a bit long and self-contained instead of splitting them in parts?

Congrats for the good work.

[From now on, back issues will be updated to contain links to the continuation articles. This will of course only work on the official AR Tech Journal WWW sites.]

From: Brian Turmelle bturmelle@ctron.com

Hello,
This is very strange. When I click on the Issue 1,April 07, 1995 archive HTML, instead of trying to download the file I get what looks like binary information on the screen. I'm using netscape but for some reason it's reading it as ascii instead of knowing to download it. I've never experienced this at any other site. Perhaps it's not correctly archived?

-

[This is an unfortunate problem in WWW servers and browsers that do not recognise an LhA archive as binary data. The easiest remedy is to turn on loading to local disk before following the archive link. This is the "load to local disk" option in Mosaic, and the "d)ownload" command in Lynx. In Netscape, click on the link while holding down the shift key.]

From: "Richard N. Hurt" rhurt@thepoint.net

Hello all,
Thanx for putting together a good Amiga programming source of info. I only have a few suggestions.

* What about a series of 'Beginning to program on the Amiga' articles? Using different languages (not just C).
* Comparison of different programming languages? Amiga-E, LISP, ARexx, etc...
* Hardware section? How to put in that extra floppy. How to autoconfig that memory.

Keep up the good work! :)

-

[I hope you like the Amiga E course starting in this issue. Any hardware experts out there interested in writing some articles?]

1.5 Programming in C

Programming in C

By Ken Howe <khowe90@entergy.com>

About three months ago I finally gained access to the internet and started browsing around Aminet. I usually look for utilities and source code to help me in my programming efforts on the Amiga. I also read Amiga Report and lo and behold I spotted a copy of the first Issue of Amiga Report Tech Journal. After reading the first and now second edition I thought the concept had great potential.

I know how difficult it is learning to program the Amiga especially as most Amiga magazines only touch the surface of programming, running the same old simple programming tutorials. I used to buy those magazines month in month out hoping to find something useful but no. So I soldiered on as most people have and learned what I know very slowly. I am by no means a great Amiga programmer and alot of people may already know most of the concepts that I will be talking about in the coming issues so if you think you can do better put together an Article and maybe you can teach me something!!

Ive been doing small development on the Amiga on and off for about 3 years mainly just to teach myself C and soon C++. I program Client Server Windows Application for a living, can Program over 10 different programming languages and have been programming for over eight years. I may not be an Amiga expert but I hope I know a bit about programming.

Well thats the background, Ive only had a week to put this article together so Im just going to do a few programming tips, mainly a few tricks I have learn't about ANSI C. I hope to do more article for future issues with a bit more meat in them!

1. Stop yourself from making the if(a = b) mistake by declaring a define statement as follows

   ```
   #define EQU ==
   ```

   Then anywhere you would normally use if(a == b) use if(a EQU b). Your C compiler may already have this defined. I always include a header file of mine called Kens.h with this define and a few other things just to make sure.

2. Initialised auto variables for strings, arrays or any composite object is very inefficient. If you do not recognise the name it simply put means a variable that is automatically initialised for you. Example:

   ```
   int i = 0;
   ```

   Each object defined in this way will occupy exactly twice the storage space. Also the compiler will reset the variable every time the function is called. This involves copying the variable contents from a safe location in memory into your working variable.

This not only wastes CPU time, it also makes for larger memory overheads and larger stack usage. In most cases the variable can be changed to be static if this is not the case then try to use a global buffer shared between all routines, which you can initialise yourself. Example:

```
Replace: void func( void ) { char ls_Msg = "This is time consuming";...}
With:    void func( void ) { static char ls_Msg = "Much Better";...}
```

3. Pre increment/decrement operators are more efficient than post increment/decrement operators. This is valid mostly when they are combined in expressions using compare. Example

   ```
   if ( ++li_A < li_B )
   ```

   There is no saving when they are being used in the simple for of li_A++ or li_B++. The reason for the benefit of (++li_A over (li_A++ is that the compiler must generate an extra jump instruction to prevent the increment from happening.

4. You can reduce the size of simple if/else blocks, where the instructions are single lines by always performing the else instruction then having a plain if. You remove a JMP instruction from your code. Example:

   ```
   Replace: if( li_A < li_B ) li_Z = 10; else li_Z = 0;
   With:    li_Z = 0; if( li_A < li_B ) li_Z = 10;
   ```

5. You can use register variables to dramatically increase the speed of loops within your program but! you must be careful when declaring more than one register in a single declare statement.

   When you declare the register variables, declare them singularly with the inner most variable declared first. Thus making sure the most used variable is the register variable whenever a spare register is available. Example:

   ```
   Replace: register int li_I, li_J;
   With:    register int li_J;  /* J in inner most variable */
            register int li_I;
   ```

6. In SAS/C you can change the size of the default console window by including the following command just after your last include:

   ```
   char __stdiowin[]="CON:0/0/700/550/Window Title";
   ```

   This command works the same as the DOS shell program so you can adjust the size and coordinates of the window.

7. Care should be taken when defining a macro that makes up more than one statement, for example:

   ```
   #define ABORT (void)printf("Aborting\n");exit(1);
   ```

   It is easy then to use ABORT in other parts of the program but it can cause problems when not used in a straight forward manner such as

in an if statement. In order to avoid problems like this you should always enclose multi statement macros in a do {} while(0) construct, for example:

```
#define ABORT do {(void)printf("Aborting\n");exit(1);} while(0);
```

8. Parameterized macros should always have parentheses () around the parameters in a parameterized macro. The helps to overcome the following.

```
#define SQR( number ) ( number * number )

li_Size = SQR( 5 )
```

equates to

```
li_Size = ( 5 * 5 )
```

but consider

```
li_Size = SQR( 5 + 6 )
```

equates to

```
li_Size = ( 5 + 6 * 5 + 6 )
```

Therefore if we use the rule of always putting parentheses around parameters we overcome the problem, for example:

```
#define SQR( number ) ( ( number ) * ( number ) )

li_Size = SQR( 5 + 6 )
```

gives

```
li_Size = ( ( 5 + 6 ) * ( 5 + 6 ) )
```

(NOTE) never use ++ and -- with macros, consider:

```
li_Size = SQR( li_Val++ )
```

equates to

```
li_Size = ( ( li_Val++ ) * ( li_Val++ ) )
```

Yuk!

9. The last tip is not much of a tip really if you are already programming in C but! if you are thinking about starting C you will probably want a book to learn from. And being new to C you will probably ask other people for advice and they will say... buy the K&R book The C programming language.

   Well I have to disagree, working in computing I am fortunate to have access to lots of books on programming and if there is a book I want to read I can order it free of charge. Most people recommend the K&R book as they have not tried many other books.

I found the K&R book very tuff going and Iam not new to programming. Therefore I would like to take this opportunity to recommend the best two book out of the 12 I have at my disposal:

As a quick reference: C Quick Reference, by QUE, ISBN 0-88022-372-3 (about $8) Is a compact (154 pages) reference to the C language and is great for looking up commands and C syntax.

As a tutorial book: Using C, by QUE, ISBN 0-88022-571-8 (about $30) Is almost 1000 pages. It is split into 3 books, 1) the tutorial is brilliant with loads of meaning full examples it start easy and will take you through most of the advanced stuff. 2) A complete C library reference. 3) An introduction to C++.

Well thats all for this issue I hope to have a full and hopefully useful routine to pick over next issue.

Ken Howe can be reached at khowe90@entergy.com

1.6 AmigaE Tutorial

AmigaE course

Sebastian Rittau <Jolly_Roger@H-Raiser.Berlinet.de>

Introduction

Some of you might know AmigaE. It is a programming language for the Amiga. It is a mix between C, Pascal and has features from other programming languages like LISP. It has also an inline assembler which gives E additional power. But E is easy to understand and that's why it is the ideal language for beginners and advanced programmers. AmigaE is best at system programming.

This course should help beginners to learn E and also gives some hints for advanced programmers. You need at least AmigaE 2.1b (the last freeware version) and for the later parts AmigaE 3.1a (registered) for this course. Both versions should be available at the AmiNet (dev/e). Additionally you need at least Kickstart and Workbench 2.04 (V37).

From AmigaE 3.0 on, a tutorial to E is included in the archive. It is another good source to learn E. But this course is more system programming oriented, that means, it will show you how to program a GUI (Graphical User Interface) and how to program a good system-conform program.

Installing

After dearchiving the archive in the directory of your choice, you should make a special Shell for AmigaE:

* start your favourite text-editor (Ed, MEmacs, GoldEd, CED, ...)
  (if you use a wordprocessor you have to save the files as ASCII!)
* load "S:Shell-Startup"

```
  * add the following lines (replace <e-dir> with the directory you
    installed E in):
    cd <e-dir>
    path bin add;
    assign EModules: Modules;
  * save the file as S:E-Startup (not S:Shell-Startup!)
  * copy the file SYS:System/Shell.info to any directory and rename it
    to E-Shell.info
  * click on the new icon and choose Icons/Information... from the
    menu
  * search for the tooltype FROM=xyz and change it FROM=S:E-Startup if
    you can't find it, click on add and type in FROM=S:E-Startup and
    press return
  * click on save

Now you can start the e-shell by double-clicking on the new icon.

Okay now make a directory for the course: type in

makedir course
cd course

in the e-shell.
 Compiling programs

To make a program you have to write it in a easy readable form in any
ASCII-Editor. All commands have a name which explains their function:

PROC
       short for Procedure

IF
       if something is true

THEN
       then do something

WriteF
       Write a sentence
       and so on...

But these form is not readable for the computer. (Try to load any
program into a textviewer. You don't see PROCs and that stuff, you
only see cryptic characters) That's why we must translate the easy
readable text (called the sourcecode or only source) into a form which
is easy readable for the computer (the program). We use the program ec
which is in the bin sub-directory of the e-dir to "compile"
(translate) the sourcecode into the program. Because ec "compiles" the
program it is called "compiler".

All sourcecodes (the files which contains the for you readable
sentences) must have the ending .e to let ec know that it is a source.
To compile the program you have to type
ec <filename of the sourcecode>
in the e-shell. The ready-to-run program will have the name of the
sourcecode without the ending .e (i.e. myprog.e will become myprog and
so on...)
```

Comments

Okay, before beginning with out first program, I want to explain you what comments are and what they are user for.

You can use comments in your sourcecode to explain the function of a program-part or to give additional information about the program. Comments are ignored by the compiler. You should use comments often to explain your program. After a few weeks you don't write on a program, you can easily forget which function did what. The same problem appears on big programs ("What was this $%"&$$" proc good for?").

Another purpose of comments is debugging (i.e. finding and deleting errors). You can easily declare the parts of the program you don't need as comments.

There are to ways to make a comment:

One way is to mark the beginning of the comment with /* and the end with */:

```
/* This is a comment - it is ignored by the compiler */
```

These comments are nested, that means, that for every /* you write there must be a matching */:

```
/* This is a /* nested */ comment */
```

but:

```
/* This would give an /* error because there are too many /*s */
```

and:

```
/* Here are too many */s */
```

These comments can have more than one line:

```
/* This comment
   is longer
   than one line */
```

The second way to make comments need AmigaE 3.0 or above. You can declare the rest of a line as comment by using -> :

```
-> This is a comment

INC x  -> This too, but INC x is a command
```

PROCs

E-Programs are built of so-called procedures. Each procedure is a little program which does a special job: a procedure could do output on the screen, square a number, make a GUI, play a sound-routine, free all memory that was allocated by another procedure and so on. Every procedure can call another procedures.

To declare a procedure:

```
PROC nameofproc()
/* Here are your commands */
ENDPROC
```

It is called like nameofproc()

PROC
declares that here a new procedure begins. PROC must be uppercase, because it is an keyword

nameofproc
you must give every procedure an unique name. The first character must be lowercase!
valid names: myProc, tHiSmYoWnPrOc
not allowed: MyProc, ThIsMyOwNpRoC

()
Every procedure can have parameters. These parameters must be enclosed by parantheses behind the name. (Parameters will be explained later) If a procedure has no parameters, the room between the parantheses remains empty

ENDPROC
ends a procedure. For every PROC there must be a matching ENDPROC

nameofproc
to call a procedure just type its name, followed by

()
Again the parameters must be enclosed by parantheses behind the name

```
 PROC main()
```

The most important procedure is called main. Every program must have a procedure with this name. The procedure main is called automatically when the program is started. If the ENDPROC of the proc main is hit, the program quits.

Okay, let's write our first program:
* open your favourite editor
* type in the following program:

```
/* My first E-program */
PROC main()
ENDPROC
```

* save it as <e-dir>/course/firstprog.e
* change to the e-shell and type ec firstprog to compile the program. If ec reports an error, check your code
* if you now type dir there should be the following output:

```
firstprog                          firstprog.e
```

* start firstprog. It is the ready-to-run-program

Whats that? Nothing happens? Oh, well, we have a proc but we have no command in this proc. So read in the next chapter about a command which writes something on the Shell.

WriteF()

With the command WriteF() you can write a string on the current shell. (For advanced users to the stdout). If no current Shell is defined, WriteF() opens a new Shell. A string is a sentence that is stored somewhere in the memory.

Simple usage: WriteF(string)

WriteF
the command

(...)
the parantheses close in the arguments for this command

string
at the moment, we only use one argument. It is the string we want to write to the Shell

Strings must be closed in by apostrophes: 'This is a string'

If you forget the apostrophes, the compiler will report an error.

Example: WriteF('Hello World!')
The sentence "Hello World!" would written on the Shell.

Okay, let's add a WriteF() to our little program: add the line

```
WriteF('Hello World!')
```

to our program. Where do you have to put the line? Well, try it out, compile the program and if the compiler returns no errors, start it.

If you really don't know what to do, the program should now look like:

```
/* My first E-program */

PROC main()
  WriteF('Hello World!')
ENDPROC
```

The indentation before the line WriteF(...) has not to be written. It is only there to make clear, what belongs to the proc main. You should use these indentations to make clear, which command-pairs belong together.

Examples: PROC-ENDPROC; WHILE-ENDWHILE; IF-ELSE-ELSEIF-ENDIF and so on.

If you have done it right, on the Shell there should be something like

```
Hello World!5.Work:AmigaE/Course>
```

Hmmmmmm, the prompt follows immediatly after "Hello World!". That

means that we have to include a linefeed (i.e. a character which begins a new line) after "Hello World!". But because a linefeed is an unprintable character, we have to use a controlcode. Controlcodes begin with a backslash ("\") followed by a character. The character "n" is used for a "newline" (i.e. a linefeed). That means that you must add "\n" at the end of the string: WriteF('Hello World!\n')

Save, compile and start the changed version. Well, now it should be correct... Wow, that is your first real program in E.

Again PROCs (Boring theory again :( )

Okay, let's write a program which uses different procs (You don't have to take over all comments to your own program, but you should take over the comments which explain the program):

```
/* Not longer my first program
   This one should show you how to use different procs */

/* Let's begin with the main-procedure
   Some people write the main-proc at the end of their programs, but I
   write them at the beginning */
PROC main()
  WriteF('Before the first proc\n') /* Don't forget the newline (\n) */
  /* Let's call another procedure. The program goes on with firstproc */
  firstproc()
  WriteF('After the first and before the second proc\n')
  /* Now let's call the last proc */
  secondproc()
  WriteF('After the second proc\n')
ENDPROC

PROC firstproc()
  WriteF('Now the program is in the first proc\n')
ENDPROC /* The program returns to the command it called */

PROC secondproc() IS WriteF('Now the program is in the second proc\n')
```

Save, compile and run it.

The output should be:

```
Before the first proc
Now the program is in the first proc
After the first and before the second proc
Now the program is in the second proc
After the second proc
```

Maybe you have noticed the line

```
PROC secondproc() IS WriteF('Now the program is in the second proc\n')
```

Well, this is a oneline-procedure. You can use these procs if you only use one command. You don't have to write ENDPROC.

PROC

define that a proc begins

```
 secondproc()
        name and arguments

 IS
        keyword that this is a oneline-proc

 WriteF(...)
        The command

 Okay, what does our program do?
   * First it writes a sentence (Before the first proc)
   * Then it jumps into the procedure "firstproc"
   * There it writes another sentence
   * Now it returns to the commands after firstproc()
   * There it writes the third sentence
   * It jumps in another proc "secondproc"
   * The fourth sentence
   * An it returns again, now to WriteF('After the second proc')
   * There it writes the last sentence
   * At last, it quits

 Now delete the line

secondproc()

 What should happen? Just try the program to know whether you guess was
 right.

 Well, now let's give you another example of procs:

/* Another proc-example */

PROC main()
  firstproc()
  secondproc()
ENDPROC

PROC firstproc()
  secondproc()
ENDPROC

PROC secondproc()
  WriteF('This sentence should appear twice!\nIt is only written in the secon d  ↩
     proc')
ENDPROC

 Next chapter it becomes really interesting. We learn about variables.
 They are very powerfull. Every program needs them!
```

About the author

Well, I'm a student, which biggest hobby is my Amiga :). I bought my first Computer (an A500) about 6 years ago and was facinated. After a short time I began to write my first small programs in AmigaBasic (Microsoft-Crap). When I bought a modem, last year, I discovered AmigaE (V2.1b) and began to learn it. It was my first real

programming-language and thatswhy I had big problems with the libraries and all that OS-stuff (I never had any book to programming on the Amiga). Well, actually I try to wriggle through Assembly.

On the computer, I mainly program and visit bbs'es, but I am also interested in much other things (like creating HTML-pages, though I have no WWW-Access). My other hobby is learning for the school (not a real hobby, but the only thing, I have time for).

1.7 Introduction to Flex and Bison

Introduction to Flex and Bison

Joe C. Solinsky <jcsky@cs.UCR.edu>

Introduction

There are a couple of languages out there that come with the GCC distribution on Aminet which most of us will overlook as some sort of enigmatic utility, unless they are pointed out. Flex and Bison are a team of languages which sit on top of C or C++, and do a lot of the dirty work in lexical analysis. If Flex is used just by itself, your program can recognize single words or perform pattern matching based on what are known as regular expressions. If Bison is used on top of that, your program can handle grammar structures to deal with an increasingly complex input, to the level of being able to write your own programming language (compiled in C, of course), if for some reason you would want to do this.

I won't pretend to explain either one of these languages to any useful level, because they are rather lengthy, but I hope that these basics will give you a feel for what is going on as you read the core of this article. Please read the extensive and useful AmigaGuide documentation which is in the GCC 2.6.3 distribution. I found it to be more useful at times in a hypertext format than the traditional books by O'Reilly & Associates, although I am not about to knock their book (which have the auspicious title of Lex & Yacc, based on the minutely different predecessors of Flex and Bison named respectively), and I admit to reading it cover-to-cover (excepting the appendices which seem to be mainly for the 'old salts' of previous lexical analyzer tools).

Flex

The input to Flex can be any standard input stream in C, including direct user input via an AmigaShell window,text file, or the output of another program (and possibly other ways of generating input that I have not thought of). In the input, ANSI characters are considered valid, so from the theory point of view, your "alphabet" is all ASCII letters, numbers, whitespace, and punctuation can be dealt with. Since the source code for these programs is available, it is possible to modify Flex to accept other symbols, and possibly with a bit of effort, binary files, but I personally wouldn't try that without extreme confidence in what was to be done.

Bison

Bison accepts the tokens that Flex can generate, which are read in

from a table of tokens generated by Bison, with the use of the %token command. It also recognizes single characters, so there is some overlap with Flex, but consider it the second stage of a two-part process, not the end-all lexical analyzer, and you will save yourself much grievance upon this inevitable conclusion. It is a really really good idea to double-check the case and spelling of the return values of Flex and the tokens that Bison recognizes, as they must be properly identified at the top of the Bison program (on top of using the standard include file bison.tab.h , which is the table I just mentioned), because technically speaking, the include file bison.tab.h is built off of the return values of the Flex program. Bison senses these tokens as integers with the use of C's #define preprocessor instruction, as you can observe in the include file, so if you chose to do something with numbers, be aware of the dreaded "magic numbers" pitfall of programming constants into your code.

Bison and Flex programs by themselves wouldn't compile under C. As I said before, they are preprocessor languages (meaning the Flex and Bison Code must be processed first), but they generate C source code (which is a typically large file, I might add, so be prepared for that) which can be compiled with your GCC compiler, or other compilers, assuming you are doing things which are pure to ANSI C, and do not reference things specific to a given compiler or Operating System (otherwise, you are stuck preprocessing on your Amiga only!). One benefit of this flexibility through standards is that should your Flex and Bison preprocessor code become lengthy, the task of turning it into ANSI C could be left up to another, more powerful machine, and with the blessings of a good modem, this machine could be anywhere, like the monster server at work or school which always seems to have spare CPU cycles for an avid programmer like yourself. This time is considerable for larger programs, and if you are doing fancy tricks, you will soon see that the preprocessing can be a considerable chunk of time on your Amiga.

The GCC AmigaGuide documentation on Flex and Bison make reference to some rather simple calculators and a Pascal language recognizer. This is primarily because these are the classic examples of what Flex and Bison are good for. The real purpose of this article is to get past that, and focus on the application of these two languages, not the instruction itself(how could I possibly top GNU documentation?). As I worked with these languages at my University, I discovered that state machines (which is the theoretical machine which they are based upon) are capable of much more, and should be utilized to their fullest. Only a modicund of work is necessary to do powerful things with these languages, and I feel it adds to the community of the Amiga if we, the programmers, show off how clever we are, to the benefit of our fellow end-users.

Anagrams

Just a week or so before I began this article, I saw a posting on comp.sys.amiga.misc which listed "Top Ten Anagrams" for the title of Amiga Technologies, the entity which brings us newer, faster Amigas from Germany. I pursued the poster of the article to find out where the anagrams came from, as I was suspicious that it was done with a computer. Sure enough, there was a WWW site which would take a string in and output as many anagrams as it could form from the string. It didn't take me long to figure out that the Flex program to do this was

conceivable, through the theoretical construct of set notation:

Set 1 is all tokens which are words in the English language.

Set 2 is all tokens in Set 1 which use the characters in the input string, but no more than once per occurence, otherwise you would end up with "too" as a valid word formed from the input "to", which is impossible, because "to" has only one ’o’ in it to begin with.

The English language is large. That’s why we made Set 2. I realize this can work for any language, but there would need to be modifications made to Flex to recognize things like umlauts and estsets, much less all of Kanji. Any language spoken by a culture is going to be computationally large. That is why the program to make these anagrams must focus itself to only valid words.

Once one has Set 2 defined, the following must occur to finally get the anagram:

Start with the largest word (sized by the number of characters in it) constructible by the input string in Set 2, then remove those letters found in that first word from the input string, like you were crossing them off an imaginary list of letters that you have used up.

Pass what’s left through the code that recognized the first English word that was found, doing this recursively.

If you use up all the letters in the original input string, then print the anagram. Otherwise, you have leftover letters, and did not succeed in making an anagram.

Permanently remove that first word from Set 2, once you have exhausted all attempts at making an anagram with the first word used first. Why? Well, you will end up with every possible order of words for every set of words which fits the anagram, and much of this is not something that needs to be done with the parser (making various orders of the words in the anagram does not involve parsing, it involves sorting), and besides, this is going to be a much larger list. I suppose this is up to you, but passing just my full name resulted in 45 different anagrams, where each anagram was a unique set of words, and had I opted to have all the pattern orders calculated, the number of responses would be roughly 45 times the factorial of the average number of words in each anagram.

After removing the first word of Set 2, pass the second largest word constructible by the input string in Set 2. If you repeat as you did above for the first, using recursion, you will ultimately get all unique anagrams that can be made from the input string.

Scanning for words

This little project is a fairly short program to write in Flex, and it would be significantly more challenging to write in C. Unfortunately, the code for this program is nowhere near as intuitive as the breakdown of the problem. Just to give you something that is a bit easier to approach in terms of solving on your own, in case this puzzle has you stumped, think of a program in Flex (only) that scans the words of the English language and returns only those which have

the letters A, E, I, O, and U occuring in that natural order (interspersed with consonants, of course). "Facetious" is one such word. The solution for this problem is quite short.

The first step is to make a single predefined set of all consonants, and then reference it inside the rules. The set looks like this (CONSONT is my name for the set):

```
CONSONT [bcdfghjklmnpqrstvwxyz]*
```

And it can be referenced in the rules section to solve the puzzle in a single line:

```
a{CONSONT}e{CONSONT}i{CONSONT}o{CONSONT}u{CONSONT}
```

Although, realistically, the puzzle needs a few more lines for housekeeping purposes, these two lines are the 'creative' part of the problem.

A complete Flex input file, with C++ main(), is included as puzzle.yy. It can be tested with the commands
flex puzzle.yy (which creates a C++ source file named lex.yy.c)
cc lex.yy.c (or sc for SAS/C, of course)

Flex is quite useful for problems that deal with formatted input. Bison, on the other hand, is better for complex structural input. The examples mentioned earlier, like those of a language parser or a calculator do indeed show that Bison can handle tasks which would be difficult in Flex, and require programming in Flex's own state machine commands, which would tend to require redundancy for every input in every state which is not relative to a valid input pattern (or collection of tokens that forms a pattern recognized by a grammar). Take a look at the examples in the AmigaGuide documentation for Bison, and see what I mean.

Uses for Flex

Perhaps you are asking yourself "why do I want to program another calculator?" or "I don't even use Pascal anymore, why would I want to parse it?" as I did when I first mulled over these examples. The answer most likely is that you are not. Flex and Bison together are tools that we as programmers can use to make our work easier (and with them, perhaps someday we can have a program which solves crossword puzzles by brute force permutation). The most common programming tool that uses these two languages is what is known as a pretty printer. Basically, this is a program that makes appropriate use of white space in our source code, and indents and flushes our code so that when reading it, the lexical levels (scope levels) appear as successive indentations, our comments line up neatly, and functions that wrap around to the next line do so neatly. But, there are pretty printers out there, and there's one in GNU Emacs that runs on the fly as you code, part of the text editor itself (written in Lisp, I believe). Lexical analyzers can be made to preprocess your source code and check for bugs in implementations of libraries, giving you feedback where your compiler misses details (or gives you nearly meaningless error codes like "parse error," which tells you your grammar seems to be off). They also appear in newer word processors (on the IBM platform, as far as I have seen) that check to see if the word you are using is

in its dictionary of the English language, and fix the spelling for you on the fly. Unfortunately, this isn't a good way to improve your spelling skills, and often, it doesn't handle the actual use of the word (their as opposed to there), leaving you with not a spelling error, but scattered English grammar. Command shells can also be built with lexical analyzers, and as we have seen with programs like KingCon and others, a lot of control is added over the standard AmigaShell.

The development tool I would like to suggest experimenting with is meant for the ambitious (for the non-ambitious, work on the word puzzles until you are ready) programmer who wants to make lexical analyzers work for him/her in the professional arena. Often times, as we are faced with developing software, there is a tremendous amount of documentation which is coupled with what needs to be written in the actual program, and comments galore. As humans, we programmers face the daunting task of remembering what var1 is, and why it is equal to my_input1 (good examples of bad variable names), and in the process of software development, we forget, again and again. We make silly mistakes because it's 3:00am and the side-effects of too much coffee are making us jittery and distracted.

Documentation and the classical techniques of software development are designed to help us, not waste our time when we could be debugging. If we were to somehow encorporate documentation with software development, and then reference the documentation as we code our software, a lot of pitfalls could effectively be removed from the software development process. The next question, naturally, is how.

The answer is totally dependent on your views, but here is my perception of things, and I hope it can spark even better ideas as you read. When I develop software, I recognize a few different stages of it. There's the analysis and specifications phase, where the idea is nailed down and copied off the white board and onto paper. Then there is the design phase, where the implementation of the idea concerns itself with issues of input devices, methoods of response, machine specifications, application appearance and operation, and other issues of the software as the user would see it. After that is established, the architecture, what is behind the design, is hammered out. Functions are given names and purposes, data structures are contrived and outlined, and program control is carefully mapped out. After the program is structured, the application then goes through a process of narrowing focus on each function, algorithm, and manner of execution, until finally there is a program sitting on the hard drive which you (and your development team) have written, and you are positive it is worth something because you tested it and it works and it is what you wanted.

Lexical analyzers can come in handy as the programs by which programs are written under. With enough flexibility, a good software designer application can address all of these phases of software development with the general principal of question and answer. Take the following interaction as a possible way that this tool could behave:

1. Ask the designer for a few sentences on what the program to be written will do.
2. Ask the designer to expand the abstract of the program when it finds certain key words

user
I want to write a video game where you are flying this ship through a jungle and shoot at aliens. The aliens fly at you in different waves, and get harder to shoot. Also, as you go along, you can pick up weapons and make your ship faster, get extra lives, and bonus points. There will be a main boss enemy at the end which is hard to kill.

design tool
So you are writing a 'game,' eh? Good idea. When you described it with 'you are flying,' what did you mean by 'you?'

user
the player.

design tool
Then it should read like this, right? 'I want to write a video game where the player is flying'

user
yes.

design tool
How about 'aliens fly at you,' is that the same 'you' as 'the player'?

user
yes.

[and so on, until every use of the word 'you' has been clarified. When the meaning of the word 'you' branches, the computer starts asking which 'you' it is based on a larger range of possibilities.]

design tool
So, as I understand it, your 'game' can be described as:
I want to write a video game where the player is flying this ship through a jungle and shoots at aliens. The aliens fly at the player in different waves, and get harder to shoot. Also, as the player goes along, the player can pick up weapons and make the player's ship faster, get extra lives, and bonus points. There will be a main boss enemy at the end which is hard to kill.

design tool
Are you happy with this definition, or do you want to make corrections?

user
It's fine. [or any variety of acceptable responses]

design tool
Alright, let's move on.

design tool
Tell me more about the 'ship' you mentioned here: 'the player is flying this ship'.

user
It’s a spaceship. It is an Alpha Class fighter.

design tool
Is this the same ship as referenced in ’make the player’s ship faster’?

user
yes.

[and so on. The idea is to expand definitions as much as possible to avoid ambiguity, and create appropriate handles for the design tool, as well as save the user from writing documentation which isn’t intuitively defined. In the end, the short description could look something like this.]

I want to write a video game where the player is flying this Alpha Class fighter-spaceship through a jungle on the planet Zarkan and shoots weapons such as missiles, lasers, and nuclear warheads at three-headed space aliens, giant snake space aliens, blood-sucking vampire aliens, and octopus aliens. The same aliens fly at the player by weaving back and forth, dancing across the edges of the screen, darting in from the corners, and materializing randomly on the screen, in different waves, where the same aliens will fly in squadrons of the different kinds of aliens and in different numbers, and it gets harder for the Alpha Class fighter-spaceship to shoot its weapons at the same aliens. Also, as the player goes along the jungle on Zartan, the player can pick up more powerful missiles, broader lasers, and more dammaging nuclear warheads, and make the Alpha Class fighter-spaceship manouver on the screen faster, get extra lives which allow the player to start where the player was killed and continue, and bonus points which increase the player’s overall score. There will be a main boss enemy alien that attacks the Alpha Class fighter-spaceship at the end of the jungle on Zartan, and the boss enemy alien is hard for the player to kill.

At this point, you are probably saying to yourself "okay kid, let’s see this wonder program of yours so far, written in something which has previously only been demonstrated as a calculator builder." Right you are, this is daunting and ambitious. Here’s how I would do it:

By establishing certain rules of the English language, we can familarize ourselves with possible ambiguities. The most obvious one is the use of pronouns in a sentence. Take this sentence as an example:

When my brother took me to see our dad, he said that mom was going to be late for dinner.

Who spoke? It is totally vague. It should logically be the brother because the brother was the subject of the time clause, and therefore has more weight than the infinitive phrase ’to see our dad,’ but it could very well be the dad who spoke. Watch for this in conversations, see how many times you have to ask whether it was the dad or the brother (metaphorically speaking) who spoke.

By first addressing this ambiguity, the specifications document can

become significantly clearer, almost as if you had someone there to proofread for you. In the English language, there are happily a finite set of words that can be classified as pronouns. It is relatively easy to parse them out and poll the user to specify what is meant at each instance, then replace the pronouns with only the object of the sentence which was typed in as a response to reference the pronoun(and parsing the English language is easy enough with Bison, because a good writing style handbook will have some notes about sentence structure, and this can be translated into something Bison can recognize). Oh, sure, this application can be fooled with enough trickery, but that's why programs come with instructions telling you not to use the subjunctive when addressing the program (or at least some mechanism in the parser which produces an error message about the grammar being too complex in the response, for the parser to understand, and poll the user again).

With this mechanism of polling responses and doing textual replacements, all the pronouns can be clarified. In addition, the implied subjects of gerunds can be specified where the structure of the sentence makes it vague, and it can also correct poor grammar (as a note, Final Copy II does grammar checking, so don't think this is unheard of).

Once things are laid out, and everyhing is undeniably under only a single interpretation when read, you will have a pretty solid document. Why go to so much trouble? Well, the original reason for this is so that people who read it don't have to ask questions like "who said mom was going to be late for dinner? I'm not sure which person is saying that." The additional reason revolves around the fact that all documentation in this application builder is functional.

The document can slowly grow into something complete, to the point where no further questions can be asked regarding details of the program. For example, at one point, I mentioned nuclear warheads in my sample run. These are considered weapons that belong to the fighter-spaceship controlled by the player. Weapons can be fired at the enemy. They can also hit the enemy and be upgraded as the game goes along.

From a structural standpoint, the application can ask How the nuclear warhead is fired by the player, how the game will know when they hit the enemy, what happens when they hit the enemy (explosion, more points, what happens to the enemy when it is hit, and so on...), and how they are upgraded (at perhaps a certain number of game points, perhaps something is dropped by the enemy when it is destroyed, you decide.).

From a visual standpoint, things can be asked like when the nuclear warheads are fired by the player from the fighter-spaceship, what does it look like? What does it sound like? Obviously, there is no room for me to expand this entirely, even from this tiny aspect of the programming, but I think the idea is clear enough.

Given the visual descriptions (which might again be polled if it uses some kind of gerundive like flashing or burning, which hints at an animation or special effect), the application can define enough of the details that this portion of the design can be distributed to the

artist, and the artist will know what to draw and how many drawings will need to be made for just this portion.

Given the structural descriptions, the application can build its own state machine for the game, determine all the modules which need to be built (ones that launch warheads, ones which handle aliens being hit by warheads, etc.), and even begin on assembling actual program control based on enough of this high-level context. In fact, if this program is used more than once, it is possible that a library can be established over time, containing code modules to handle frequent events, and given the hard drive space, this sort of application could truly blossom into a powerhouse of application building.

The example of the video game seems a bit complex. Games are complex. But, I wanted to show the flexibility that has to be considered for a project like this. Try to consider a simple data entry program with a graphical interface. The documentation for this is fairly short, assuming it is primitive. There's some entry functions, access functions, program control, file I/O, and maybe printing facilities. There are gadgets and menu options on a single window. There is likely little artwork or sound effects, no animations, and no musical score (unless you are weird). Still, it is asking a lot of our minds to constantly have a clear picture of the entire application that doesn't change or get confused or forget parts. If we do a layout of all the buttons and text fields, what happens if somewhere down the line, we forget to add a button? The further down the line, the more code is lost, and the greater chance that there will be a bug. Right?

The actual specifications for this application builder are somewhat beyond the range of a document meant to inspire programmers into using Flex an Bison in new and innovative ways, and I appologize for not being able to do this, but like the anagram problem, the implementation is tremendous, because details simply are not covered.

I hope you can appreciate, as I have, the value in these small programs, and use them to their fullest potential. However, I have one frightening word of caution about all of this. GNU is part of the Free Software Foundation. You are well advised to read the legal preambles to the use of Flex and Bison. You may discover that your application is subject to being only freeware, much to the chagrin of your monumental intentions. I believe there is a point where significant modifications made on the source code generated by Flex and Bison will result in it no longer being under these rules, however, if you are thinking of doing a search and replace on variable names, moving functions around, and a few other tricks, just remember what these programs are designed to do, and think to yourself that GNU probably has some sort of way of testing for typical changes to their code. There is a light at the end of this tunnel, though. The source code for these applications is freely distributable, and so modifications of that source code and compilations of your own de-FSF-ized versions of Flex and Bison could result in a nullification of GNU's hold over what you write with their programs. If you are serious, talk to a lawyer, and don't take the word of a college student (me) on what the law states.

A simple resolve to this (assuming my last theory holds in a court of law) might be to make an implementation of Flex and Bison that makes

specific calls to the Amiga libraries, and not the ANSI C text I/O functions. Whatever the case, just remember to check with your lawyer before putting a pricetag on it.

If you have read this entire document and are still at a loss as to what Flex and Bison can do for you, here are some ideas worth exploring:

Write a program that scans in AmigaBasic programs and converts them to C source code, so all your Basic code can have a second life.

Write a hard disk organizer which reads in all of your .info files and does such things as intuitively assign your favorite text reader as the default tool to all the text files, or your image viewer for image files. How many times have you gotten an archive off the net with lots of little text files, only to face the fact that you have to edit each .info files yourself? Wouldn't it be nice if something were to run and look for references to known text editors (and their paths) and reassign them to what you actually have?

Write a C parser which inputs your old source code and makes an attempt at optimizing readiblity by modularizing your 1000 line functions into smaller pieces, then placing them in separate source files.

Write a text encryption program however you wish.

Write a program that builds form letters based on a core letter, variable points, and a separate file formatted with those variables (people's names, addresses, etc.).

If you have to distribute source code with your application and you don't want people to understand it, write your own program which muddles the readability of your code by stripping comments, using incremental nondescript variable names, and replacing text segments with long, tedious strings of ASCII hex codes.

Write a program which makes a feeble attempt at commenting Assembly code on the Amiga, by identifying what certain base adresses represent in memory operations or library calls, cite what the value of certain registers are (based on where they were originally input) and what they change to), and generate a separate program flow chart, perhaps in the form of a real picture.

Write a floppy-disk organizer for floppy-based systems which stratifies individual disks based on file content (a disk full of all your letters to mom, another with your collection of sound effects), and optimizes space usage to fill each disk to the brim, or to whatever capacity you wish to have remaining, then swaps around all your files for you.

Please don't let your own list be this list alone. It shouldn't stop here. I encourage you to talk with your fellow Amiga programming peers and divas about solutions to some of these problems. In these uncertain times, the best thing we can do for the machine we love is make programs which don't exist anywhere else, and pioneer high-tech, clever software which the end user has just got to have, because it is

too cool to do without. If you have questions about Flex or Bison, are having trouble installing your free GCC compiler, or you need some kind of informed response to a question that covers the issues I mentioned, feel free to ask. I will do my best to answer as my own work schedule allows. But, before I get barraged with questions about shift-reduce and reduce-reduce errors, let me say this much: when you are using these languages, implement the technique of stepwise refinement; get a small part of it to work concretely, then add to that portion. It is tempting to lay things out all at once, pray, and compile, but when you have 100 lines of Bison rules, and over 400 reduce-reduce conflicts, it really does take less time to start over, using your rules as a template to stepwise refinement. You will learn the (poorly explained anywhere you look) fickleness of token look-aheads and impossible or vague grammars if you deal with your errors one at a time. And remember, even though there may be hundreds of conflicts, there may be only one error in your grammar.

About the author

Joe C. Solinsky was a student at the University of California at Riverside for 4 years. He is currently doing research at the University in robot simulators as a graphics programmer, but is looking to move out of the Ivory Tower of college, and into the real world again. As an Amiga user since 1987, he is die-hard about his favorite machine. He can be reached the following ways:
jcsky@cs.ucr.edu
joe@mindesign.com
2442 Iowa Avenue Apartment K-8
Riverside, CA 92507
USA
(909) 788-5408

1.8 Installer

Installer Basics

Robert Reiswig <rcr@netcom.com>

What version of the Installer are you using?

This time I am taking a bit of a break from doing the Play16 installer example. I wanted to talk abit about the different versions of the Installer. As I mentioned in my second article, two different divisions of Commodore contracted out to Sylvan Technical Arts for the installer. CATS contracted then to do an installer for Kickstart 1.3 and 2.0, while the contract by Engineering was for 2.0 only.

As a result, there are quite a few different versions of the Installer out there. The problem is that they all do not support the same commands. Though 95% of all the Installer scripts you run across will work with just about any version of the Installer binary, it seems that a few of the more complex and bigger PD packages take full advantage of all that it has to offer. This includes using some of the commands that are not supported by some of the different versions of

the Installer.

The new X-Windows package for the Amiga, AmiWin, uses the commodore Installer. He uses the BITAND command and this seems not to be supported in many version of the installer. (AmiWin20d.lha on Aminet in gfx/x11) The CyberGraphX installer uses the 'newname' option for a 'copyfiles', this also seems to have problems on some installers.
Versions

Here is a list of Installer versions that I have collected over time. If you have a version that I am missing, if you could please uuencode it and email it to me at installer@vgr.com that would be great! To check to see what verison of the installer you are running get into a cli/shell and type: 'version installer full' . If you installer is Imploded with Imploder then you will not be able to get a version string from the cli/shell. You will need to run the installer with a script and select the 'About' button.

Imploder is a program that 'packs' or compress executables and still leaves them in a runable format. The big advantage is that it saves room. When you put together a distribution disk and you are running out of room many people use Imploder to implode the installer binary and cut the size in about half. Imploder can be found on Aminet as: imploder-4.0.lzh in the util/pack directory. Even Commodore sent disks out with the installer binary Imploded.

The imploded sizes in the following table are after using Imploder 4.0 with the default settings.

Version String		Size	Imploded
installer 1.24	(1-09-92)	115144	62684
installer_2 2.9	(23-03-92)	111504	60056
installer_2 2.12	(15-05-92)	113224	60936
installer_2 2.15	(3-08-92)	113740	61300
installer_2 2.17	(13-02-93)	113532	61236

What Version to use?

What version should you use? Well if you can get version 2.17 off an Commodore Install disk that might be best. 2.17 seems to do everything. The next best bet it to grab Installer-1.25.lzh in util/misc of aminet. This archive really has version 1.24 in it. This is what most Commercial programs come with and seems to run just fine. Also 1.24 can be found on Fred Fish 870.
NOTE

If you do not already have the "Installer" program in your "c:" directory, it would be a good idea to get your Install Disk (the one that comes with the Operating System Disks), and copy the Installer from there to "c:" _or_ at least place it somewhere in your path.

Also pointed out in the last issue by the Editor (something that I should have done) is that you _should not_ hardcode a path to the installer in the tooltypes. Instead just have 'installer' for your 'Default Tool' in the installer script icon.

Well that is about it for this installment ... hoped you learned

something!! If you wish to get more detailed information there is an Installer archive in/on the Fred Fish Collection that covers the _older_ 1.24 installer.

1.9 ARexx tutorial

Arexx Tutorial

By Josef Faulkner (panther@gate.net)

1.0 - Introduction

I am assuming you know what Arexx is and the history behind it. If you don't know, this information is readily available in the first chapter of the Arexx User's Guide that comes with Workbench. This book is also a very handy reference, so dont lose it! :)

If you are familiar with any programming language from Basic to C, you will have no problem picking up Arexx. Arexx is one of the easiest languages to learn and to work with. When writing arexx programs, you do not need to deal with silly line numbers (as in basic), or worry that the computer will crash if it doesnt correctly open a library (as in C). Arexx is very difficult to crash on its own, so feel free to explore it.

Arexx is one of the things that makes the Amiga one of the best Personal Computers anyone can own. The Amiga was designed to multitask in such a way that even operating systems built 10 years later still have yet to fully close the gap on the efficiency and integration of the Amiga multitasking environment. Arexx takes advantage of the Amiga's ability to multitask by allowing us to greatly customize our favorite applications. However, it is not necessary to write arexx scripts for programs, but to write an entire program in Arexx, independent of any applications. For example:

```
/* calc.rexx - A simple command line calculator */
parse arg args
interpret 'say 'args
exit
/* End */
```

Don't worry about what this says yet, I plan to explain all of this later. To use this script, cut it out, and save it as rexx:calc.rexx. Be sure to include the comment at the top. To use it, type rx calc 2+2 in the CLI. This will also be a helpful debugging tool later on.

1.1 - Making sure everything is in place

Arexx needs a few things in place in order for it to work. Workbench

handles most of this, but it seems to have left out some important details.

1. Getting the Rexx: directory straight.
   + Make a directory on your SYS: partition named Rexx:
   + Edit your S:Startup-Sequence and change the line that says Assign >NIL: REXX: S: to Assign >NIL: REXX: SYS:Rexx
2. Making sure RexxMast is run.
   + Add this line to S:User-Startup: Run >NIL: SYS:System/Rexxmast >NIL:
3. Making sure RexxC is in the path
   + If you are running WB2.1, I dont think RexxC is put in the path on installation, so you will have to add it in your S:Startup-Sequence.
   + Look in your S:Startup-Sequence file for your path statement, and add RexxC: after SYS:Utilities/.
   + I also recommend adding REXX: to (end of) the path, so that you can later protect your arexx scripts with the script bit, and run them like a regular AmigaDOS command.
4. File check
   + Be sure the following files are in your SYS:RexxC/ directory: TS TCC RXC TCO WaitForPort HI RXSET RX RXLIB TE

1.2 - Usage

There are two ways you can run an Arexx command. One is to use the RX command located in the SYS:RexxC directory, the other is to use the script file flag to tell AmigaDOS that the script is a command.
Examples:

```
1> rx calc.rexx 2+2
4
1>

1> protect rexx:calc.rexx +S
1> calc.rexx 2+2
4
1>
```

The second method requires that REXX: is added to your path.

1.3 - Comments

All arexx script files must begin with a comment. This comment tells the arexx interpreter, as well as AmigaDOS with +S files, that this is an Arexx script.
Example:

```
/* This is a comment */
```

It doesnt matter what you put in the comment, however it is standard practice to at least include a short description of what the program does. However, try to minimize long comments at the beginning of frequently used scripts, as it might slow down the execution of the script. It is best to put long descriptions and such at the end of the arexx script, so that it isnt scanned by the interpreter.

Unlike C, comments may be nested in Arexx, although I dont see much use for it :)

1.4 - Variables

All variables in Arexx are stored as strings. However, when numerical

operations are done, Arexx will automatically treat a string holding numbers to a number, do the operation, then store the result back in a string. This has great advantages, since you wont have to do any casting, and you can mix numbers with strings very easily.

For example, in lower level languages, if you wanted a user to be able to enter either a string or a number into a variable, you have to read their input into a string, then if that string is a number, convert it to a number variable. In arexx, you simply take the input, and then you can immediately use it however you wish. Example:

```
/* ( variable.rexx ) - Shows how variables can be used */
variable='TEXT'
say variable
variable=variable' '15   /* Puts the string "15" as another word */
say variable
say word(variable,2)*2   /* word() is a function that returns a specified */
                         /* word from a string.  In this case we are */
                         /* looking at the second word, which is 15. */
                         /* This will say 30 as the answer since we are */
                         /* multiplying the second word of variable by 2 */
exit
```

Concatenation is joining two strings together. When we put the 15 in variable, we told arexx to leave a space so that 15 would be the second word. If you do not want a space, you can join two strings using the || operator (two pipes).

Appendix A

Arithmetic Operators

Char	Operation	Pri
**	Exponentiation	7
/	Division	6
%	Integer Division	6
//	Modulo (Remainder)	6
+	Addition	5
-	Subtraction	5

Comparison Operators

Char	Operation
==	Exact equality
~==	Exact Inequality
=	Equality
~=	Inequality
>	Greater Than
>=	Greater Than or Equal To -.
~<	Not Less Than -'-- Same thing
<	Less Than
<=	Less Than or Equal To -.
~>	Not Greater Than -'-- Same thing

Logical Operators

```
Char Type Pri
---- ---- ---
 ~   NOT   8
 &   AND   2
 |   OR    1
 ^   XOR   1

Other Characters

Char Description
 :   Procedure/Function Label
 ()  Group operators and operands to override priorites
 ;   Statement Terminator
 ,   Marks a continuation of a line (to a new line)
```

Written exclusively for Amiga Report Technical Journal

1.10 Results of the ARTech survey

Results of the user preference survey

In ARTech issue 2, I asked you, the readers, what aspects you feel are the most important in a good program. I got 31 answers, far too few to really count, but it might give some idea of the things users expect regardless.

Below are the choices, listed in order of importance. The score is the average score in the replies, with the range of 1 to 5.

```
 1. Stability                                      [4.97]
 2. Speed                                          [4.00]
 3. Detailed documentation                         [3.71]
 4. Extensive configurability                      [3.68]
 5. ARexx interface                                [3.58]
 6. Pretty interface                               [3.55]
 7. Interface in your native language              [3.32]
 8. Command line usage                             [3.10]
 9. Low memory usage                               [2.97]
10. No mouse required for most important features  [2.94]
11. Documentation in your native language          [2.90]
12. Many flashy features                           [2.39]
13. No mouse required for full interface control   [2.10]
```

8 of the respondents were from USA, 5 from England and Italy, 4 from Germany, 3 from Sweden, and 1 from Finland, Australia, Netherlands, Poland and Canada each.

The majority of people, 20, found ARTech on Aminet, with WWW and BBS's both getting 5 people. One person got ARTech from Compuserve.

1.11 BOOPSI guide

BOOPSI Overview - part 2

By Chris Aldi <caldi@pcnet.com>

This article is designed to provide the novice BOOPSI programmer information to aid in writing a custom gadget class for an application. The reader is assumed to be familiar with some basic BOOPSI and/or OOP concepts. I recommend reading the BOOPSI chapter of the RKM:Libraries Manual, 3rd Edition, then perhaps come back and read this overview again and it should start to make more sence. Also, if you missed it the first part of this article appeared in the premier issue of AR Tech.

In part one, the generic BOOPSI object methods were discussed, as well as the gadget methods specific to gadgetclass writers. Now, we will take a quick look at some of the imageclass methods, discuss some rendering optimizations which you can apply to both gadget or image classes, and provide some example source code which can be used as a basis for custom class work.

Image objects

Just as all gadgetclass objects are of the struct Gadget type, image class objects are of the struct Image type. The superclass will fill out a good portion of the Image or Gadget structures at OM_NEW or OM_SET time based on the tag attributes passed via SetAttrs(), SetGadgetAttrs() or NewObject() function calls.

Image methods

Method spefic message structure definitions can be found in intuition/imageclass.h, note the first ULONG of the message will be the MethodID idenfier which your class dispatcher can use to direct the message to the proper function in your image class.

The follow methods are common for all classes of image objects:

IM_HITTEST

For rectangular images all you have to do is pass this method to your superclass which will handle it. If you so choose to handle this method yourself, you muse return TRUE if the point is within your image structure's TopEdge, LeftEdge, Width, Height fields. The IM_HITTEST method uses a impHitTest message structure.

IM_HITFRAME

This method is very simular to IM_HITTEST, but is a special version for images that support IM_DRAWFRAME. It is just like IM_HITTEST except it tests if the point is would be within the image if the image is clipped or scaled to some rectangular bounds. It too uses a

impHitTest structure. Take note, imageclass treats IM_HITFRAME exactly like IM_HITTEST, thus ignoring the restricting dimmensions.

IM_ERASE

Intuition will send this method when an image object is erased with the EraseImage() function. This method, like IM_HITTEST, can be passed to the superclass. The imageclass will use EraseRect() to erase the image using the values in the object's image structure. It uses a impErase message structure.

There doesn't appear to be an IM_ERASEFRAME method, so if your object supports IM_DRAWFRAME be sure that imageclass or your class handles IM_ERASE appropriatly for your image object so not to erase anything outside of the last rendered clipping/scaling IM_DRAWFRAME dimmensions.

IM_DRAW

This method is sent when an image object needs to rendered. Typically the application or perhaps a gadgetclass will use DrawImageState() to render an image. For boopsi images, this will cause an IM_DRAW method to occur. This method uses the impDraw message structure. Contained in this structure is an imp_State field which defines which state the image should be rendered in. Some common states are IDS_NORMAL, IDS_SELECTED (typically, you would reverse shine/shadow pens for bevels, perhaps show some alternate image, etc), IDS_DISABLED (render image with a ghosting pattern over it), etc.

Generally you can get your pen settings from this structure as well by looking at imp_DrInfo->dri_Pens but imp_DrInfo can be NULL in some cases, so your advised to check for this and optionally use some reasonable default pen settings.

IM_DRAWFRAME

This method is much like IM_DRAW except it uses an extended message structure which includes some bounds clipping dimmensions. At your option, your image class should scale or clip rendering to stay within this "frame" area.

RENDERING TIPS & SUGGESTIONS

1. For complex imagery with a great deal of rastport setting changes such as pen colors, fill modes, etc, use cloned rastports. This is simple enough to do and doesn't use all that much memory either. For example:

```
struct RastPort *rp;
struct RastPort clone_rp1;
struct RastPort clone_rp2;

clone_rp1 = *impDraw->imp_RPort;
clone_rp2 = *impDraw->imp_RPort;

SetAPen(&clone_rp1, impDraw->DrInfo->dri_Pens[FILLPEN]);
SetAPen(&clone_rp2, impDraw->DrInfo->dri_Pens[TEXTPEN]);
```

Now, you can render in TEXTPEN via clone_rp2, and in FILLPEN via clone_rp1. Since SetAPen() and simular functions often involve some complex computations, there is something to be gained by pre-computing the rastport settings and using the appropreatly set rastport.

2. Take care not to render in the same place twice. By this I mean when filling the area inside a button, do not over render the bevel box then re-render a surrounding bevel. This will cause unwanted flashing of the bevels. Also, do not assume the bevel boxes are of a fixed size unless they are your own. Bevels provided by frameiclass could change thickness in a future OS revision, as could something like ClassAct's bevel.image change its bevel size/style via revisions or use preferences programs. It is always best to query the beveling image for its bevel bar size, and account for that in any rendering operations you do.

3. When rendering, do as little work as possible. When rendering gadget imagery, do not render imagery that has not changed. For example, a scroller gadget need only render the arrow button that was selected and again when deselected. It should never need to re-render any bevel around the proportional slider area, or render the other arrow button. When dragging the proportional slider, there is no reason to re-render the arrows at all. Since the gadget knows what element of the scroller made it active, you can set some flag variables or bitmask in GM_GOACTIVE, them evaluate them in GM_RENDER when the redraw method is GREDRAW_UPDATE. When the redraw method is GREDRAW_REDRAW a complete refresh is needed regardless of the flag settings. You can determin this by looking at the gpRender structure's gpr_Redraw field (see intuition/gadgetclass.h).

Chris Aldi is the co-founder and president of Phantom Development, and a co-author of the ClassAct BOOPSI toolkit. You can reach him at caldi@pcnet.com.

1.12 Contributors, staff, and contact addresses

Amiga Report Technical Journal is an FS Publications production.

Staff

Producing Editor, Osma Ahvenlampi, Osma.Ahvenlampi@hut.fi

Supervising Editor, Jason Compton, jcompton@xnet.com, FS Publications.

Contributors to this issue

Sebastian Rittau, Jolly_Roger@H-Raiser.Berlinet.de

Joe C. Solinsky, jcsky@cs.UCR.edu

Ken Howe, khowe90@entergy.com

Chris Aldi, caldi@pcnet.com

Robert Reiswig, rcr@netcom.com

Josef Faulkner, panther@gate.net

Contact addresses

Email: artech@warped.com

Snail mail:
ARTech c/o Osma Ahvenlampi
Rekipellontie 2 F 55
00940 Helsinki
Finland