The Electronic Magazine for Macintosh™ Developers

Issue #1

5/3/85

Contents

This issue uses the following fonts: New York 9, 10, 12, and 18 point; and Geneva 9 and 10 point. If you are going to print the magazine, these fonts and the sizes twice as large should be present.

MacDeveloper is published the first Friday of each month. Distribution is via electronic bulletin board systems and national information services. If these avenues of distribution are not accessible to you, send a self-addressed, stamped envelope with a Macintosh diskette to the following address; unless you request a different issue, the latest issue of MacDeveloper will be returned.

Harry Chesley
1850 Union Street, #360
San Francisco, CA 94123

Special notice to BBS operators: You are actively encouraged to post this magazine on your BBS, so long as all of it is posted. The larger the distribution of MacDeveloper, the more articles and advertising we will get to support the magazine, to everyone's benefit.

Editorial: Electronic Publishing

There have been a number of articles recently proclaiming a new age of publishing. All of this, they say, is due to the Macintosh's ability to handle both text and graphics, multiple fonts, and high quality output devices such as the LaserWriter, and even certain photo-typesetters. Now, they say, it is possible to publish a magazine or newsletter for a fraction of what it once cost, at least in the areas of editing and page composition.

Indeed, a number of publications are already being produced using the Macintosh. And the results are spectacular. Apple's **Outside Macintosh**, **The Club Mac News**, and many local user group newsletters are all starting out as LaserWriter output. And it's truly hard to tell the difference between them and magazines produced with traditional methods.

MacDeveloper is intended to help take this progress one step further.

For the same reasons that the Macintosh is an excellent machine on which to compose a newsletter, it is also an excellent machine on which to read one. Why do we need to print, reproduce, and distribute information on paper, when we can keep it entirely in the form of electrons and magnetic fluxes? And even if you do want your magazine on paper (easier to take to bed to read), with the Mac you can still print it out at your end — or even better, print out only those parts you specifically want to read. If you have access to a LaserWriter, you can get a printed copy as good as many printed magazines; and the ImageWriter ain't that shabby either.

This is certainly not the first time that people have published information electronically, either entirely or in part (some magazines have an on-line supplement to a traditional printed version). But the advent of the Macintosh does mark the first time you can publish high quality text and graphics this way, the first time you can match traditional methods in every aspect except color.

Neither is it true that all the barriers in the way of electronic publishing have been removed. A very small percentage of potential readers have the equipment needed to access the electronic distribution services. Even fewer have the interest and knowledge to do so. Transmission speeds are still quite slow, thus limiting the possible size of the publications. Means for economically supporting the publications are not entirely clear — this magazine is intended to be advertiser supported, but some types of publications cannot be supported by advertisers without compromising their purposes.

Problems like these are to be expected in the early period of any new activity, even more so when new technology is also involved.

Another source of problems is that the technology may be **too** successful at allowing inexpensive publishing. What do you do when everyone publishes, or at least can publish their own newsletter? Which do you read? The deluge of information may become overwhelming.

But I like to think that we can use better metrics for the value of a piece of information than how glossy the paper or how high the resolution of the graphics.

How to Scroll

Larry Rosenstein, Apple Computer

Introduction

An important part of the Macintosh user interface is scrolling. Most applications implement scrolling since, in general, an entire document cannot be displayed on the screen at one time. The Macintosh user interface guidelines specifies how scrolling should appear to the user. The implementation of scrolling, however, is up to the individual developer.

This article describes an implementation of scrolling that can be used in most applications. The reader should be familiar with the Control Manager, Quickdraw, and Window Manager sections of *Inside Macintosh*; it might be helpful to have a copy of these sections handy while reading the rest of the article. The program fragments in the article are given in Pascal, but can be easily adapted to other languages.

The case we will consider is that of a window with both horizontal and vertical scroll bars that displays only one kind of data at any time. Windows such as those used in *MacDraw*, which display both a palette and a scrollable drawing area, are simple extensions of this case.

Definitions

There are a couple of definitions that will make it easier to describe the scrolling implementation.

The first is *view*. The view is the image of the data that appears in the window. In a spreadsheet program, the view would contain the values in all the cells of the spreadsheet as well as the horizontal and vertical grid lines. At any point in time, the view has a definite size and some part of it is displayed in the window.

In this implementation, it doesn't matter what is displayed in the view. The only requirement is that there is a procedure DrawView that can be used to draw the view on request. (Note that this requirement means that there must be some internal data structure, which DrawView uses to do its job.)

There is a Quickdraw-style coordinate system attached to the view, called the *view coordinate system*. By convention, the point (0, 0) is at the top left corner of the view coordinate system. We can also define a *window coordinate system*, which is fixed to the window; (0, 0) in this system is always at the top left corner of the window's content rectangle.

Finally, there is the concept of the *scroll position*. This is the point in view coordinates that appears at (0, 0) in the window coordinate system.

A Scrolling Implementation

There are several aspects to the implementation of scrolling. First, the application must be able to draw the contents of the window in response to an update event. This will be done by the procedure DoUpdate, which looks like this:

```
BEGIN
BeginUpdate(theWindow);
SetPort(theWindow);

(* draw the scroll bars *)

(* draw the correct part of the view *)

EndUpdate(theWindow);
END;
```

This is just the standard skeleton for handling update events.

The scroll bars are controls; they are drawn using the Control Manager procedure DrawControls. The view is drawn using the DrawView procedure mentioned above. Before calling either of these procedures, we must ensure that the window's origin and clipping are set up properly.

The scroll bars are fixed within the window; it is convenient to position them relative to the window coordinate system. Therefore, before calling DrawControls the origin must be set to (0, 0). The clipping must be large enough to include both scroll bars. It is easiest to just set the clipping to the window's entire content rectangle.

The part of the view that is displayed in the window, however, varies with the scroll position. One way to accomplish this is to have DrawView take the scroll position into account when it does its drawing. There is an easier way, however; we can set the window's origin to the scroll position before calling DrawView. If we set the origin of the window to (x, y), then the point (x, y) will appear at the top left corner of the window, which is exactly the definition of the scroll position.

MacDeveloper

The clipping for the view must be set to the window's content rectangle minus the scroll bars. Standard scroll bars are 16 pixels wide (or high) and they overlap the edge of the window by 1 pixel. To calculate the clipping rectangle, therefore, we start with the window's content rectangle and subtract 15 from the right and bottom sides.

After these additions, the DoUpdate procedure looks like this:

```
BEGIN
BeginUpdate(theWindow);
SetPort(theWindow);

(* draw the scroll bars *)
SetOrigin(0, 0);
ClipRect(thePort^.portRect);
DrawControls(theWindow);

(* draw the correct part of the view *)
SetOrigin(scrollPosition.h, scrollPosition.v);
r := thePort^.portRect;
r.right := r.right - 15;
r.bottom := r.bottom - 15;
ClipRect(r);
DrawView;

EndUpdate(theWindow);
END;
```

It is important to remember that the SetOrigin procedure does not adjust the clipping rectangle. That is why DoUpdate sets the origin before the clipping.

Now that we have a procedure for drawing the window, we can start to manage the scroll bars.

We will make the convention that the scroll bars' control values specify the *desired* scroll position. The desired scroll position may be different from the current scroll position, which is stored in the scrollPosition variable. To reconcile this difference we will use the procedure DoScrolling.

DoScrolling must compute the difference between the current and desired scroll positions, scroll the view within the window to bring these values into agreement, and update the necessary parts of the window. For doing the actual scrolling, we will use the Quickdraw procedure ScrollRect. DoScrolling, therefore, looks like this:

```
BEGIN
(* compute the desired scroll position *)
SetPt(desired,          GetCtlValue(horzontalScrollBar),
                        GetCtlValue(verticalScrollBar));

(* compute the amount to scroll by in delta *)
delta := scrollPosition;
SubPt(desired, delta);

(* setup the clipping to exclude the scroll bars *)
r := thePort^.portRect;
r.right := r.right - 15;
r.bottom := r.bottom - 15;
ClipRect(r);

(*  scroll  *)
ScrollRect(r, delta.h, delta.v, invalidRgn);

(* mark the invalid region and update it *)
InvalRgn(invalidRgn);
DoUpdate;

(* remember the new scroll position *)
scrollPosition := desired;
END;
```

(One possible optimization of the above code is to check for delta being (0, 0) before calling ScrollRect, etc.) We don't need to set the origin in the above piece of code, because we take the current origin, regardless of what it is, into account when we use thePort^.portRect.

The last part of the scrolling implementation is the handling of the user's mouse clicks.

When the user presses the mouse button, the application receives a mouse down event. The first step in processing a mouse down event is to call the procedures FindWindow and FindControl to determine in what part of the window the user clicked. If the user did click in a scroll bar, then FindControl will return both the scroll bar and a *part code* indicating where in the scroll bar the user clicked.

There are 2 different cases to consider: (1) the user clicks in one of the scroll arrows or one of the gray areas and (2) the user clicks in the scroll box. In both cases, we call the Control Manager procedure TrackControl, to process the mouse click.

If the click is not in the scroll box, then we want to scroll the document while the mouse button is held down. To do this we pass an *action procedure* to TrackControl, called TrackScroll. TrackScroll takes 2 parameters: theControl and partCode, both of which are returned by FindControl.

TrackScroll first determines the distance to scroll by, based on the orientation of the scroll bar (horizontal or vertical) and the part code. If the user is scrolling towards the start of the view, then this distance should be negative. The orientation of the scroll bar is determined by its longer side.

TrackScroll then adjusts the scroll bar's control value by the scrolling distance, and calls DoScrolling to make the scrolling happen on the screen. Since the action procedure is called repeatedly while the mouse button is held down, the view will continually scroll until the button is released.

If the user clicked in the scroll box, we don't do any scrolling until the mouse button is released. Therefore, we pass TrackControl NIL as the action procedure. The Control Manager will follow the mouse and set the scroll bar's control value when the button is released; when TrackControl returns, we just call DoScrolling to make the change visible on the screen.

Putting this all together, we get the following procedure for handling mouse clicks in a scroll bar (the parameters are theControl, partCode, and mousePoint):

```
BEGIN
SetOrigin(0, 0);
GlobalToLocal(mousePoint); (* convert to local coordinates *)

CASE partCode OF
    inUpButton, inDownButton, inPageUp, inPageDown:
        partCode := TrackControl(theControl, mousePoint,
@TrackScroll);
    inThumb:
        BEGIN
        partCode := TrackControl(theControl, mousePoint, NIL);
        DoScrolling;
        END;
END;
```

Since the scroll bars are positioned within the window coordinate system, it is important that the window's origin is set to (0, 0) before calling any Control Manager routines. Also, remember that when a mouse down event is received, the mouse coordinates are in global (screen) coordinates, while TrackControl expects local (window) coordinates.

The procedure TrackScroll has parameters theControl and partCode, and looks like this:

```
BEGIN
IF partCode <> 0 THEN (* ensure that the mouse is still in the original
part *)
    BEGIN
    delta := <distance to scroll by>;

    (* change the scroll bar to the desired scroll position *)
    SetCtlValue(theControl, GetCtlValue(theControl) + delta);

    (* make the change on the screen *)
    DoScrolling;

    (* set up the origin as expected by the Control Manager *)
    SetOrigin(0, 0);
    END;
END;
```

The final SetOrigin is necessary because DoScrolling calls DoUpdate, which changes the window's origin. (If you leave it out, your program will not continually scroll while the mouse button is held down.)

Enhancements

The preceding section described a general scrolling implementation. In this section, I mention a couple of enhancements that are also useful to include in your application.

First, it is not necessary that DrawView draw the entire view every time. The DrawView procedure can do some optimizations by examining the window's visRgn and computing from that what parts of the view need to be redrawn. This optimization is especially important for scrolling because (1) usually only a small rectangular part of the view needs to be redrawn and (2) the update time will directly impact the speed of scrolling.

Second, the implementation of DoScrolling does not depend on the code that calls TrackControl. In fact we can implement automatic scrolling by setting the scroll bars' control values and calling DoScrolling. Automatic scrolling is necessary for things like searches in a word processor (when the target is found we want to scroll it into view). The Macintosh user interface guidelines mention other cases in which automatic scrolling is important.

One thing that hasn't been mentioned yet is setting the scroll bars' maximum and minimum values. Since the scroll bar control value represents the desired scroll position, the maximum and minimum should be set to the maximum and minimum

scroll position.

The minimum scroll position is just the start of the view, which we defined to be 0. The maximum is not the size of the view, however, since that setting would allow the user to scroll the entire view out of the window. Instead, we must subtract the size of the window from the size of the view, and set the scroll bar maximum to that value.

It is possible that the view is smaller than the window; that is, the entire view fits within the window. In that case the calculation for the scroll bar maximum would be negative. We must check for this case and set the maximum to 0, in order to ensure that the maximum is not less than the minimum.

This check has another benefit. If the entire view fits within the window, we set both the maximum and minimum to 0, which disables the scroll bar. This is desirable because it informs the user that everything in the document is visible. (If you examine the Finder, you will see that it follows this user interface.)

Since the scroll bar maximum depends on both the size of the view and the size of the window, we must recalculate it if either of these changes. Also, setting the maximum could affect the current value (the Control Manager always ensures that the current value is between the minimum and maximum), so we have to call DoScrolling after setting the maximum.

Finally, there is one change to TrackScroll that improves the aesthetics. We could check that the user has reached the end of the scrolling range before calling SetCtlValue and DoScrolling. The Control Manager always redraws the scroll box when SetCtlValue is called, even if the value does not change. By doing this test, we will prevent the scroll box from flashing when it reaches the end of the scroll bar.

A Review of the Aztec C Compiler for the Macintosh

Harry R. Chesley

Introduction

The C language is one of the best tools for developing new programs on the Macintosh, or on any other machine. If you agree with that statement, read on. If not, it is beyond the scope of this article to convince you.

Aztec C, by Manx Software Systems, is one of the leading C compilers for the Macintosh. Others include Consulair C and MegaMax C — the MegaMax compiler is actually too new to thoroughly evaluate, but it looks like a strong contender.

This article reviews the Aztec C compiler for the Macintosh. It does not describe the language itself; readers unfamiliar with C should read *The C Programming Language*, by Kernighan and Ritchie. It does describe the compiler, the environment it runs in, the other tools supplied by Manx with the compiler, and it makes a quick stab at performance analysis.

Manx Software Systems, Inc.

Manx Software Systems has been producing C compilers for a long time. I originally became aware of them while evaluating C compilers for use with a CP/M-80 system some years ago — they came in first overall and second for our application, which had some unusual requirements.

Manx has compilers and/or cross-compilers (a cross-compiler runs on one machine but produces code for another) for:

- PCDOS & MSDOS machines
- Apple II
- Commodore 64
- CP/M-86 machines
- CP/M-80 machines

- TRS-80, models III and 4

It's possible, therefore, that Manx may have a compiler which would allow you to develop Macintosh code from another system, or to use the Macintosh to develop code for some different computer.

This review, however, concentrates on the Manx Macintosh compiler running on the Macintosh.

Hardware Requirements

Aztec C will run on a minimum configuration Macintosh: a 128K machine with no external disk drive. For serious development, however, I recommend a 512K Mac with the external drive and an ImageWriter printer. This is the configuration that was used for evaluating the compiler.

The Compiler

The Aztec C compiler handles a very complete version of C, including both single and double precision floating point (float & double), typedefs, and register variables. It does not include bit fields, enumerated types, or structure passing.

It uses 32-bit pointers, so it can address the full Macintosh memory (both the 512K now available, and also any amount to be added in the future). Note, however, that this means that some previously written C code may not port easily, since ints and pointers are not the same size. On the other hand, the programmer who wrote that code should not have assumed that they were the same size... This is not a problem with the Manx compiler, but a historical anomaly stemming from the fact that C started on a 16-bit machine (the PDP-11).

The following table shows how Aztec C implements various types:

Type	Size	Comments
char	8 bits	Can be signed or unsigned; signed by default
int or short	16 bits	
long	32 bits	
float	32 bits	IEEE format
double	64 bits	IEEE format
pointer	32 bits	

The compiler converts the source program to assembly language, which must then be assembled using the assembler supplied with the system. The MDS assembler is not used, and you are not required to buy the MDS separately. The assembler that is used is a full 68000 macro assembler. It is more than sufficient for handling the output of the compiler plus occasional small pieces of assembly source code. The main limitation of the assembler is the lack of assembly level Toolbox/OS support. This is discussed in a subsequent section on Toolbox support.

MacDeveloper

The compiler produces quite efficient code. It runs the ten iteration Sieve benchmark in 3.8 seconds (with register variables), for those who follow benchmark numbers (for those who don't, the PDP-11/40 C compiler was clocked at 6.10 seconds, and 8MHz 68000 assembly language at 1.12 seconds). Benchmarks, however, are of questionable value — if nothing else, since they've been quoted so widely compiler writers are now optimizing the compilers for those test cases, which doesn't necessarily mean that the compiler will perform as well on the code you want compiled. More significant, therefore, is Manx's long history of compilers. They have the expertise and experience to "do it right."

One example of this is that the compiler provides register variables: four data and two address registers (out of the 68000's eight address and eight data registers) can be used to hold variables of the user's choice. Since many functions only have this number or fewer variables (assuming modular code), it is often possible to keep all local variables in registers. This can dramatically decrease execution times, with very little work on the part of the programmer. The following piece of code, for instance, took 134 seconds without the register declaration, but only 86 seconds as shown below:

```
register long l;

for (l = 0; l < 10000000; l++);
```

Another example is the sieve benchmark, which with register variables took 3.8 seconds, but without them took 6.5 seconds.

(NOTE: This is a very slim treatment of the quality of the code produced by the compiler. It would be useful to have an article comparing this and other compilers in more detail in this area.)

Linking and Resources

On the Macintosh, producing the final, runable file involves linking object code modules together with a linker and including resources generated independently of the C code.

The Manx linker combines two types of files: object files and libraries. The difference between the two is that all of an object file will be included in the linked file, while libraries will be searched to resolve external references, and only selected portions will be included in the linked file. Therefore, if you don't use the sin() function, it won't take up space in your final, runable file.

There are two libraries provided with the compiler: Lib.c, which includes run-time support required by the compiler, some of the Toolbox/OS interface, miscellaneous run-time utility functions, and a non-floating point version of printf()/scanf(). Lib.m includes the transcendental functions, such as sin(), plus floating point

versions of printf() and scanf().

The linker will optionally produce application files to be run under the Shell (where the console I/O is performed by the Shell) or under the Finder. It can also be used to produce desk accessories.

For dealing with resources, Manx distributes Apple's RMaker program, which allows you to compile resource descriptions, and to combine the resources with code files produced by the linker. They are also planning to distribute the Resource Editor when a released version of it becomes available.

Toolbox/Operating System Support

A complete set of "#include files" is provided to interface to the Macintosh ToolBox and Operating System. These files define all the Toolbox/OS structures your program might use, along with the appropriate constants.

For calling Toolbox or OS routines, the compiler can be told that a given function is a Pascal function (which implies a different parameter passing technique), and what trap instruction to execute to call it. Declarations of this type for Toolbox and OS functions are supplied in the "#include files" mentioned above. For most cases, therefore, the compiler will generate in-line code for Toolbox/OS trap calls. In a limited number of cases, an intermediary function in the run-time library is called.

After several months of use, I've found a few definitions to be missing, and one or two to be wrong. When I reported the wrong ones to Manx, they were already aware of them and said they would be corrected in the next update (more on updates below). Considering the size of the Toolbox/OS interface, this isn't bad.

On the other hand, there is no support at all given to assembly language access to the Toolbox/OS. Since most C programmers use assembly only in limited areas, where C just won't do for reasons of speed, etc., this is not a major limitation. But Aztec C can't be considered both a C and assembly language product for the Macintosh. It is a C compiler with some assembly language support as well.

The Development Environment

The compiler does not run under the standard Apple Finder. Instead it uses a program called the Shell (Unix terminology). This program provides a user interface which is very similar to Unix and very dissimilar to Apple's Finder. Unix programmers will feel very much at home in this environment. Finder fanatics will hate it. This is so emotional an issue that I'm going to defer it to a later section called "Why I Don't Hate the Shell."

In addition to providing the primary user interface, the Shell also handles copy protection. It will not run (and since you must run the compiler from the Shell, neither will the compile run), until you insert a key disk. This must be done once at the beginning of each session. It does not, however, need to be re-inserted every time a program is run, even if you leave the Shell and then come back to it. Two key disks

are supplied, so you do get a back-up key. Although I'm not very fond of copy protection, this is the least offensive approach that I know of, since you only have to think about it once per programming session — which for me generally means once a day.

Besides the Shell, compiler, assembler, and linker, Manx provides a number of useful utility programs, including:

Z: This editor is very similar to the Berkeley Unix editor Vi. Once again, if you're used to Berkeley Unix, you'll feel right at home. If not, skip this one (hint: it doesn't use the mouse at all).

Edit: This is the Apple editor from the MDS, supplied as a part of the Manx package. It is a good, multiple window editor for programming. (It isn't for word processing, if nothing else, because you're restricted to one font type per file; no switching in the middle.)

RMaker: As mentioned above, this is Apple's resource compiler.

Make: This is one carry-over from Unix that everyone should appreciate. It allows you to describe which source, object, and executable files depend on which other files. For instance, foo.o (object) might depend on foo.c and foo.h (source), and foo (runable) might depend on foo.o and foolib.lib. Once the "makefile" describing these dependencies is created, Make will check the "last modified" dates on all the files and decide which should be rebuilt, then execute the commands needed to rebuild them. Make is a great program for two reasons: (1) it automates the building process so you can go get something to eat while its compiling, linking, etc.; and (2) you don't have to worry about whether you remembered to recompile all the parts you changed: Make will insure that everything is really, definitely up to date.

Grep: Another good program from Unix, Grep will search a file for any lines with a given string of characters in them. The string can be fixed or include "wildcard" parts; for instance, "grep m..x prog.c" will display all the lines in the file prog.c which contain a four character sequence beginning with "m" and ending with "x".

Diff: This program compares two files and displays the differences. It's very useful when you've lost track of which version is which, or you've just gotten an update to a source file and want to know what's changed.

MacsBug: This is the standard Apple debugger. The Manx Shell can also be told to capture error traps and return to the Shell with a register dump on the screen.

RamDisk: The RAM disk supplied is a simple but reliable program that takes most of the memory in the 512K Mac and makes it a very fast disk.

I haven't described all of the programs that come with the Manx system (there's also an archive program, for instance), but I think this gives you the idea that it's a very complete development environment.

MacDeveloper

In addition to the utilities, the system comes with several examples, some from Apple, some from Manx. These include:

- term: a terminal emulation program.
- explor: a desk accessory written in C.
- grow: the C version of the Pascal program in Inside Macintosh.

The source code for the RAM disk and several of the utility programs are also supplied.

Documentation, Upgrades, and Technical Support

Aztec C comes with two three-ring binders full of documentation, covering the operation of the Shell, the compiler, linker, and assembler, utility programs such as Make and Grep, descriptions of the run-time library (printf(), disk I/O, etc.), and descriptions of the Toolbox/OS interface.

There is no index, so you sometimes have to search for a while, and some of the more obscure Toolbox and Operating System interface details are not documented. But besides these omissions, everything that you need is there, and then some. It's not intended to be a substitute for Inside Macintosh, which is still required to do real Mac programming, but it provides everything you need to know about the Manx system.

Since buying the compiler in October, I've received two complete updates, with another one expected in the near future. The updates consist of a set of disks — which must be returned in order to receive the next update — containing new copies of everything (compiler, assembler, linker, Shell, utilities, examples, etc.), plus an addendum to the manual describing the contents of the disks and the changes since the last release, plus any new or updated documentation.

On the occasions that I called for technical support, they have been friendly and helpful. I hear from others that they've had the same type of experience with Manx technical support.

One year of updates and technical telephone support is included with the compiler.

Why I Don't Hate the Shell

I've used Unix systems for years, so I may have found it less painful to use the Manx Shell than some people will, but given that caveat, I'd like to argue that the Shell is not a major step backward, even if it isn't the Finder.

First, let me describe the way I've configured my system: I have all the frequently used programs on a RAM disk — these include the Shell, Edit, Make, the compiler, assembler, and linker, and RMaker — the operating system and less frequently used

programs on one disk drive, and the source code under development on another drive.

Because they're on the RAM disk, and because the Shell is much faster than the Finder, I can switch between programs **very** quickly. And, of course, I have a whole disk free for the program I'm developing.

I spend almost all of my time in one of two places: Edit or Make. Edit is the same editor as is provided with the MDS, so it looks very Mac-like. And with Make, I just start it up and let run.

As a result, I have an environment which is very fast, which for the most important parts follows the Mac user interface, but which also gives me the best of the Unix-style environment when I want it.

I still wish Manx would give some thought to integrating their product into the MDS development environment, but I'm not exactly suffering the way it is now.

Summary

The main drawbacks of the Aztec C compiler and development environment are:

- Does not use the Finder — for some people, this is an insurmountable objection.
- Some limitations in documentation, and no index.
- The price: $499 (a more limited version is available for $299).

The main advantages are:

- Full, fast implementation of C.
- Full Toolbox and Operating System support.
- Useful utility programs also supplied (Make, Grep, Edit, etc.).
- Good support.

For my money, and quite a bit of it too, Aztec C is an excellent compiler and development environment. I have been using it for several months, and have been very satisfied with its operations, its quality, and the code produced.

Program Launching on the Macintosh
a burned-in viewpoint

Mike Lehman,
President, Tardis Software.

When I sat down to write FastFinder, I did so because I was so frustrated with the Apple Finder's inability to deal with a large number of files on any given disk. I knew I had two initial problems: reading the disk directory and launching a program. The minimal functionality that my new Finder had to have was to display the directory and launch another program. Directory display was relatively easy, launching, on the other hand, was puzzling. The segment loader documentation describes something called the applications parameter handle. It fails to mention how to create one, access one from assembler, and many other things. I shall now enlighten you based upon my experience (at least as far as I needed to go):

- At location 000002E0 is the name of the program to load when an exit-to-shell trap is executed. It is in Pascal string format, and is normally 'FINDER'. You can change this to be any string you want. There is a limit (I think of 15 characters: 1 length + 15 data).

- At location 00000AEC is where YOU put a HANDLE to the application parameter area you must allocate on the SYSTEM heap before you execute the Launch trap. The formats described in Inside Mac are correct. In summary, 0AEC points to a pointer which points to a variable size entry array of parameter file names, their drive refnum and a print flag for each one. The list must reside in the SYSTEM heap to be available to the next program loaded since the segment loader frees the application heap on each launch (NO, I haven't figured out how to use Chain either).

The only other goodie related to launching is that the Apple Finder, before launching looks at the disk containing the application and if that disk contains a SYSTEM file it changes the BootDisk location in low memory 00000210 to the new drive and does an InitResources before the launch. This has the effect of changing the "system residence" to the new disk, thereby causing all fonts, overlays, etc. to be read from the new disk. This is good and bad. It's good if you have fonts over on the "new" disk

you need. It is bad if the new disk is a floppy and the old one is your hard disk. We decided for FastFinder not to automatically switch. You decide what is best for you.

Note: THE APPLE FINDER "TRASHES" LOCATIONS 00000000 - 00000003 WITH THE CHARACTER STRING 'FOBJ' WHICH IT TURNS OUT IS REQUIRED FOR THE ROM'S MEMORY MANAGER TO RUN. BEFORE WE TRASHED THESE LOCATIONS TO THAT VALUE IN FASTFINDER, SOME PROGRAMS, LIKE THINKTANK 512 WOULD NOT WORK. I CAN'T REALLY BELIEVE THIS FIX BUT IT SEEMS TO WORK.

Good launching to you.

Outside Outside Macintosh

The following articles are reprinted from **Outside Macintosh**, Apple's Newsletter for Certified Developers, with permission from Apple.

◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊

Unique Signatures, Fonts, File Types, and Faith

By Bill Dawson

Since my very first days with Macintosh I have been in charge of maintaining a database of Unique Signatures, File Types, and Font Numbers. The only information I had to start with was *Inside Macintosh*. Somewhere in the Structure of a Macintosh Application reads the line, "Every application must have a unique signature by which the Finder can identify it." It goes on to explain that this can be any four-character sequence and that it must be assigned by Macintosh Technical Support. Well.. I don't *assign* anything.

Instead I have been keeping track of whatever four-character sequence that any developer may want to use. This has been working out very well. Most of the time people can ship their product *as is* with the signature that they were using in the development stages. Some use strings that 'mean something' to them or have the program/company name somehow built in. And still some go for the real 'four-letter words.' Please don't get your heart set on one of these — nearly all of the good ones ARE taken! So.. if you are at that stage in development where you're going to need to keep separate any currently mounted apps, then give me a call at (408) 973-3400.

File Types

The same thing goes with File Types. To quote *Inside Macintosh* again, "Signatures work together with File Types to enable the user to open or print a document from the Finder. When the application creates a file, it sets the file's creator (unique signature) and file type. When the user asks the Finder to open or print a file, the Finder starts up the application whose signature is the file's creator and passes the

file type to the application along with other identifying information, such as the file name."

What does this mean? It means that if you don't register your File Types and Creators with me *and* you mount it on a disk with the *same* file type *and* they are incompatible *and* you call me to complain that you have garbage all over your screen (assuming that it comes up at all) I well either laugh or pretend that something is wrong with my phone or transfer you to Dial-a-Prayer.

Name That Font

This brings me to *The Dreaded Font ID Number*. Fonts as things exist now are kept separate in the Finder by Font ID Numbers (New York is 2, Geneva is 3, Monaco is 4, ...). Everything up to 255 works. What is wrong with this system? There are more than 255 fonts! Oops. So what do we do now? Fortunately all of these fonts have names. So we can keep track of them by NAME. If you have had any dealings with fonts up to this point, you know that they still need a Number to be identified by the Finder. I will continue to assign Font ID Numbers. The change will be that the number I assign you will have already been assigned to someone else — we're wrapping around. I will however, tell you who else has that font # and what the name of it is so you can warn your potential users of installing it on a disk with the same ID. Until further notice this is how the world of Fonts will be.

(Editor's note: If you expect to use a number of Signatures or File Types, you can also ask to be assigned a block of them, such as "XXXn", where n is a digit from 0 to 9.)

◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊

Software Training: An Opportunity

By Bud Colligan,
International Product Manager

The focus of our international marketing efforts for the balance of 1985 is to sell SOLUTIONS, i.e. hardware and software that TOGETHER constitute a meaningful solution for the user. One important aspect of solution selling is software training.

We have been cooperating with various third party software companies to train the Apple sales force and dealer sales people worldwide. These efforts have included training on Microsoft Word and File, Living Videotext's ThinkTank 512K, and Peachtree's Back to Basics in Canada, Australia, and the Far East. In Europe, we have focused on Odesta Helix and MacLion.

In both cases, the countries requested applications training to improve their solution selling.

Apple's international marketing groups responded by contacting software companies and requesting free software and training materials. We then customized the materials and demos into an integrated presentation for the countries. Demos emphasized real solutions, like inventory control on a database, mail merge with a word processor and project management with MacProject. Finally, we went "on the road (or plane)" with the training.

The results of our first wave of solutions training has exceeded our expectations! In Australia, we trained *200* dealer sales people in the first two weeks of March, providing local training in Sydney, Melbourne, Canberra Brisbane, Perth, Adelaide, Darwin and Townsville. In addition to software training, we took the opportunity to re-energize them on the Macintosh philosophy and product plans. Typical comments on the course evaluation form were: "This training has just reinforced my commitment to sell Macs to a corporate market, and it is great to see such a great company with a great product really stand behind and push it into the marketplace. Well done Apple...Let's show IBM who is the leader."

After reading this, you're probably wondering, "how can I sign-up?" Well, it's fairly easy. We are willing to distribute training disks and materials to all of our international areas for you. If you would like to take us up on this, please send 25 copies each to Didier Diaz (408) 973-3789 (Europe) and Jeff Galvin (408) 973-3903 (Rest of World) for distribution. (Note: We have a personnel change in the International Group. The new contact for copies of Redit, the Localizer, and localization technical support is Gabriella Martino, (408) 973-6140, Mail Stop 3L. Merein Cremer has accepted a new position within Apple.)

If the countries request a Cupertino led hand-on training for your products, we will customize demos and a training class, provided you are willing to give a complimentary copy of the software to each RSP we train. After training over 700 people in the last month, it is clear that receiving the software at the end of the training cements the enthusiasm and new knowledge of the product. Give us a call to discuss training on your product!

◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊

Apple Color Plotter Libraries Now Available

By Paul Norris

Could your application benefit from very high quality graphics output and color? Then you should consider licensing the Apple Color Plotter Libraries from Apple Licensing. The libraries are provided in source code format. To use them, you only need compile them into your program. They are called in the same way that you call Quickdraw routines; only the output is directed to the Apple Color Plotter. This is an

excellent way to provide high quality graphics to those on a limited budget. The Apple Color Plotter costs less than $800.

◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊

Nibble Mac

By David Szetela, *Nibble* Magazine

Nibble magazine's April issue will launch a regular new section devoted to the Macintosh. "Nibble Mac" will be a separate "mini-magazine" within each issue, and will contain product reviews and news releases; tips on using commercial software; and complete programs for home finance, entertainment, education, and software development. The pilot issue will feature a comparative review of the two Macintosh BASICs, a complete graphics/strategy game written in Microsoft™ BASIC, and a review of Mark of the Unicorn's Mouse Stampede™.

Products submitted for review should be sent to the attention of David Krathwohl, and press releases and new product announcements should be sent to Sharon Stentiford. *Nibble* is also soliciting programs and articles for publication in the "Nibble Mac" section. Programs should be written in Microsoft BASIC, Apple's MacBASIC, Pascal, or 68000 assembly language, and should run on a Macintosh 128K. Authors will receive royalties based on the magazine's sales of the programs on disk. For authors guidelines, write to *Nibble*, 45 Winthrop Street, Concord, MA 10742. For more information, call (617) 371-1660.

Author's Guidelines

Topics

MacDeveloper is oriented toward working Macintosh developers. Articles should speak to a reader knowledgeable about the Macintosh user interface, and at least rudimentally aware of the format and usage of the ToolBox and Operating System. Many readers will have much more knowledge and expertise than this.

Articles about development tools, how to use the Macintosh ROM, programming techniques and algorithms (in all areas), and news are welcome. Articles by companies about their own products are also welcome: MacDeveloper is intended to spread information, not necessarily to be a "consumer advocate." Permission to reprint articles from other publications is also appreciated.

Format

The newsletter is initially being published in MacWrite version 2.2 format. When the new disk-based MacWrite is widely available, that format will be used instead — this may be as early as the second issue. Articles can be submitted in either format, but non-2.2 format articles may not be published until a later issue if the switch-over has not yet been made.

The magazine consists of a number of individual files, packed together with the program PackIt. This, together with the use of MacWrite 2.2, makes it readable by a minimum configuration Macintosh. PackIt is also used to allow non-MacWrite documents, and possibly even small applications to be distributed with the magazine.

MacDeveloper uses the following fonts: Article headings are in 18-point bold New York. Section headings are in 12-point underlined New York. Normal text is 10-point New York, with full justification; 9-point New York can be used for SMALL CAPITALS, and for footnotes. Programs examples are in 10-point Geneva unless column alignment is important; then 10-point Monaco is used.

The reader will be expected to have available the fonts listed above. Others can be used, but they may not be present when the article is being read, and therefore may not look as nice.

Writers are encouraged to include MacPaint or MacDraw pictures in articles. MacDraw is preferred, since it will look much better if the article is printed on a

LaserWriter. MacPaint documents can also be submitted, but not MacDraw (too few people could read them).

Articles will be edited to incorporate standard MacDeveloper headers and footers. Additional, content editing may also be done unless the author requests otherwise; in this case, additional time may be needed to coordinate possible changes with the author.

Submission

Articles can be submitted (in order of preference) to:

(1) The MacFido BBS: 415-563-2491. Use the MACMAG file area (#9), and
upload using either normal XModem or ASCII transfer with a .HQX file, or MacBinary.

(2) By US Snail to:
Harry Chesley
1850 Union Street, #360
San Francisco, CA 94123
(Include a self-addressed, stamped envelope if you want the disk back.)

(3) By electronic mail to:
CompuServe: 71505,1711
Delphi: Chesley
The Source: BDC134
MCI Mail: HChesley
(Be sure to Binhex the file first, or else send ASCII text.)

Please include an address or phone number (preferably an electronic mail address) where I can reach you in case there are questions or problems. Unless otherwise indicated, I will assume you don't mind your work being edited. If you do mind, let me know.

DEADLINE FOR SUBMISSIONS IS TWO WEEKS BEFORE THE PUBLICATION DATE!

Advertising

For the first three issues, advertising is being published for free. After this, advertising charges will be used to support the magazine.

Ads are interspersed with the articles, with the ads matching the article topic as much as is possible (i.e., an article about a C compiler might include ads for the same compiler). The ads are offset from the articles in such a way as to clearly indicate that they are advertising.

For best effect, an ad should either be a full page or fill a single screen. Graphics can make a tremendous difference in the appearance and effectiveness of the ad.