The Electronic Magazine for Macintosh™ Developers

Issue #5

11/1/85

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 in no way sponsored by or associated with Apple Computer, Inc.

MacDeveloper is published the first Friday of every other 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 we will get to support the magazine, to everyone's benefit.

In This Issue

Several changes come to MacDeveloper starting with this issue.

First and foremost, it gives up the pretention of being a money-making venture. It is now entirely for the fun it it, and for the fun of programming the Mac.

Second, we are starting to reprint articles from user group newsletters — in this issue, an article from the Yale Macintosh User's Group. Any user group newsletter editors who read this, send in your newsletters and permission to reprint. If there are any good developer-oriented articles, we'll reprint them along with your user group name and address. Should be good for membership!

Third, a text edition of MacDeveloper is being compiled, sans font changes and graphics. This edition will be available on-line on several major timesharing systems, and should make MacDeveloper accessible to those potential readers who don't want to or can't download the MacWrite version.

Fourth, the MacWrite version is being split into files small enough to fit in a 128K Macintosh. The old issues are also being re-released split in the same way.

In this issue, we have a review of the Megamax C compiler — thanks to YMUG's DeskTop Journal. We have now reviewed Aztec, Consulair, and Megamax C, which we feel are the major players in the Macintosh C compiler arena. Note, however, that these companies, as well as other current and future competitors, are always improving their products. So check with them before concluding that a given product is missing that one key feature you've just got to have.

The "How to..." series continues with an article on using icons, certainly a central feature of the Macintosh.

And there are the usual reprints from Outside Macintosh. Many thanks to Apple for letting us reprint this very useful information — not everyone who programs the Mac is a certified developer...

Finally, I must, as usual, end with a plea for articles. There are never enough. In fact, there are seldom any at all. So, put mouse to screen and send in your Mac programming wisdom.

Review: Megamax C

Eric Celeste

This article is reprinted from The DeskTop Journal, The Newsletter of the Yale Macintosh User's Group (YMUG), with permission from YMUG. To join YMUG and subscribe to The DeskTop Journal, write to:

YMUG Membership
220 Yale Station
New Haven, CT 06520

What is "C"?

Do you want to be able to write applications for the Macintosh that anybody could just double-click from the desktop and use? Would you like to add your own special tool to the desk accessories hiding under the Apple-menu on your Macintosh? Would you like access to all the treasures stored in your Mac's 64K ROM? If you can program in C (or are willing to learn), *Megamax C* may be just what you are looking for.

Before I begin, let me make clear that I am not an expert in programming with C, nor am I knowledgeable about all the C compilers available for the Macintosh. I have only been working with C for six months, and only two of those on the Macintosh. But I do feel I've had enough exposure to the Macintosh, C programming, and the product and people of Megamax to recommend *Megamax C* to those who wish to write applications for the Mac on a Mac.

C is a structured programming language similar to Pascal. In general, programming in C leaves you a little more flexibility with variable type definitions while programming in Pascal is a little more likely to keep you from inadvertently stepping on your machine's toes (and, with a Macintosh, getting a system error). C is also somewhat less wordy than Pascal, for example the "begin" and "end" statements found in Pascal are often replaced by simple "{" and "}" symbols in C, and C usually complains less about extra semi-colons here and there. In general if you feel comfortable with Pascal, you should have no trouble picking up C with a few weeks practice.

Megamax C is compiled. This means that you have to go through a number of steps to get a working program. You must create the source code with a text editor, compile it into object code with a compiler, link the object code into executable (binary) code, and only then can you try your program. The advantage is that the resulting program will be very fast, quite compact, and totally independent of the compiler once it's finished. In other words, once you've got your program working you can copy it anywhere and pass it on to any friends or sell it as you wish. It does not need the Megamax package to run, and

Megamax does not charge any license fee for software developed with their compiler.

What You Get

Megamax C arrives on two disks. One (MM1) holds the editor, compiler, linker, header files, and system library. The other (MM2) holds a code improver, a librarian, a disassembler, Apple's resource compiler, a conversion utility, and some sample programs (including a desk accessory). (A bonus on the MM1 disk is an incredibly small system file; it only takes up 46K!) A useful desk accessory, called Transfer, facilitates jumping from program to program without going to the Finder. This saves a lot of time when moving from step to step in the edit-compile-link-execute cycle.

The editor supplied is the common Consulair editor found in many development systems. It allows multiple files to be open simultaneously and can be set to indent automatically. Overall it is a good editor, but it is an awfully big program, taking up 55K of disk space. If you don't like it you can replace it with any text editor that produces standard text files.

The *Megamax C* compiler is the highlight of this package. It presents you with a dialog box that lets you choose any file whose name ends in ".c" to be compiled. Once you've chosen your file and clicked the "compile" button it works at a blinding pace. It completes the compiling process after a single pass through your source code, and stores the object code created in a file with the same name as the original except the ".c" is replaced with a ".o". Any errors found while compiling are printed on the screen and stored in a file called "errors.out" which you can then look at as you correct your code. The offending line of code is listed along with very clear error messages.

The linker also puts up a dialog box for you to work with. To link object files you first select the ".o" files you wish to link together. Then you fill in the name you want the executable file to have and also tell the linker what filetype to give it. (The filetype option will usually be filled in with "APPL" for applications, but it is very easy to create disk accessories by filling this field with "DRVR".) Once everything is set the way you want it, click the "OK" button, and away it goes. Linking is not quite as speedy as compiling, but it is not wasting your time. Megamax calls its linker a "smart linker" because it makes sure to link only routines that the program actually needs to run. Any good linker should do this, but you pay for it with a little time.

If your program uses editable resources, you also have to use Apple's RMaker (a resource compiler) to add those resources to your executable code. One beauty of the Megamax linker is that you can recompile and link a modified version of your code to a file that has already had its resources compiled without those resources. The only thing the compiler will change is the data branch of your file.

The *Megamax C* package also includes a number of programs which I have not explored in great depth. I don't have any idea what "mmimp", the code improver, does to a program, except that it is supposed to make it more efficient. The librarian, "mmlib", lets you add

or remove files from the "syslib" which is automatically linked to all programs. The disassembler disassembles object code to, I presume, aid in debugging; since I do not know 68000 assembly language, I can't comment further. One last program included is a utility to convert the names Megamax uses for Macintosh ROM routines to and from the standard *Inside Macintosh* capitalization conventions. More about this later.

Features

Megamax C is fully "Kernighan & Ritchie" compatible. For those of you new to C, Kernighan & Ritchie are authors of a book, *The C Programming Language*, which serves as the current definition of standard C. Everything in K&R is in *Megamax C*, including full floating point number support. *Megamax C* also expands on standard C by supporting register variables, in line assembly code, and dynamic overlays which allow you to write a program bigger than available memory space by specifying which segments to load from disk at what time.

When programming on a Mac, what you really need is access to the Macintosh ROM routines. *Megamax C* provides full access to these routines through a series of header files organized around *Inside Macintosh* chapters which you can #INCLUDE in your program. A header file is a separately stored bunch of source code, which can be made a part of your program through the #INCLUDE command. The Megamax folks did not stop with just the ROM routines, as far as I can tell they have also been very thorough in defining all constants mentioned in *Inside Macintosh*.

This is also an area where those at Megamax have taken some liberties with Apple's nomenclature. The names Apple uses for the ROM routines and constants in *Inside Macintosh* are full of mid-word capitalization. Since C is case-sensitive and since it is much easier to miss-type a word like "GetNewWindow" than "getnewwindow", the Megamax routine and constant names are all fully lowercase. (The words are the same, just the capitalization changes.) I like this arrangement, since I find it much easier to type the lowercase versions. But, for those who don't like the Megamax way or who need to use a program already typed with the standard capitalization, the "convert" utility mentioned above will decapitalize or recapitalize the ROM routine names in any given source code.

One last note on Macintosh ROM routines. The ROM routines generally expect string parameters to be passed as Pascal strings, which differ slightly from C strings. With *Megamax C* you don't have to worry about this difference, the conversions are all done automatically.

Finally, an important feature of *Megamax C* is the friendliness of the Megamax people. Once you have their compiler it is yours. You never have to pay them any royalties for the programs you develop with it. If you find a possible bug in the compiler, their tech people will talk with you (and probably point out the mistake you missed!). They plan (though I haven't seen it yet) a quarterly newsletter for users, which will, among other things, keep users posted on updates. All in all, they've been very helpful and supportive.

In Summary

The *Megamax C* package runs on a 128K Macintosh with one disk drive. I used it in this way for a few weeks. It is, however, much easier to use with two drives. Most of the time you only need to use the MM1 disk, but this disk leaves very little room for working copies of your programs. A separate document disk comes in very handy unless you really enjoy swapping disks. It should be perfectly comparable with a 512K Mac, though I've never tried.

There may be other C compilers out there which are as good. Like I said above, I haven't shopped around much. But I doubt there are any which are better. *Megamax C* lets you work in a Macintosh environment building Macintosh programs. So far I have really enjoyed working with it and I highly recommend it to anyone looking for a solid software development system.

The Cost

Megamax C lists for $299.95. This makes it one of the cheaper full development systems on the market.

[Eric Celeste is a student at Yale and member of YMUG happily drowning in his Mac.]

MacDeveloper

How to Use Icons

Harry R. Chesley

1. Introduction

Icons are one of the most visually prominent aspects of the Macintosh. As soon as the machine is turned on, the user sees a disk icon with a flashing question-mark. The Finder, which is the first program most people learn to use, is primarily involved in manipulating icons (and thus the disks, files, and folders they represent).

Every final application for the Macintosh has at least one icon: the one that represents it on the Finder desk-top. Some applications use icons as a part of the user interface while the application is running. Others use icons as a convenient way to perform certain graphic operations.

This article begins with a discussion of how Macintosh icons are stored. Then it talks about how to use icons within an application. And finally, it discusses how to inform the Finder about which icons to use to represent the application and any document types it may use.

(Note: In this article I break with the tradition of using Pascal examples. Since it seems that more readers use C than Pascal, myself included, all examples are given in Aztec C.)

2. How Icons Are Stored

Macintosh icons are 32x32 dot graphic images. There's nothing magic about 32x32. Other systems have used 64x64; cursors on the Macintosh — which sometimes serve a similar function — are 16x16. 32x32 is just a convenient size, both in terms of the size on the screen and the amount of disk and RAM storage taken up.

The 32x32 bit image takes 128 bytes. It's organized exactly as you'd expect: four bytes per line, earlier bytes corresponding to left-hand blocks of dots, highest order bit representing the left-most dot in each byte:

Hex	Binary
00000000	00000000000000000000000000000000
07FFFFE0	00000111111111111111111111100000
08000010	00001000000000000000000000010000
08000010	00001000000000000000000000010000
08FFFF10	00001000111111111111111100010000
09000090	00001001000000000000000010010000
09555090	00001001010101010101000010010000
09000090	00001001000000000000000010010000
09550090	00001001010101010000000010010000
09000090	00001001000000000000000010010000
09500090	00001001010100000000000010010000
09000090	00001001000000000000000010010000
09500090	00001001010100000000000010010000
09000090	00001001000000000000000010010000
09540090	00001001010101000000000010010000
09000090	00001001000000000000000010010000
09000090	00001001000000000000000010010000
08FFFF10	00001000111111111111111100010000
08000010	00001000000000000000000000010000
08000010	00001000000000000000000000010000
08000010	00001000000000000000000000010000
08000010	00001000000000000000000000010000
0800FF10	00001000000000001111111100010000
08000010	00001000000000000000000000010000
08000010	00001000000000000000000000010000
08000010	00001000000000000000000000010000
07FFFFE0	00000111111111111111111111100000
04000020	00000100000000000000000000100000
04000020	00000100000000000000000000100000
04000020	00000100000000000000000000100000
07FFFFE0	00000111111111111111111111100000

Image

Icons are stored as 'ICON' type resources.

Sometimes, however, you want to overlay an icon on top of an existing display. In this case, you don't want to simply erase a 32x32 dot portion of the display and replace it with the icon — after all, the icon might actually be smaller than 32x32, or at least have rounded corners. Neither do you want to let what was there before "show through" — this might fill in white interior areas of your icon, making it difficult or impossible to identify.

The solution to this problem is to use a "mask." A mask is a second 32x32 dot pattern which shows which dots in the icon are actually wanted and which dots are "outside" the icon. Before the icon is drawn, those parts of the underlying display which fall under

the mask are erased. Then the icon itself can be drawn.

Another way to view masks is to consider the corresponding dots of the icon, mask, and existing display as being parameters to a function which chooses the value of the resulting dot in the new display. There are 256 possible functions of this type. For icons and masks, we use the following:

Icon	Mask	Display	New Display
On	On	On	On
On	On	Off	On
On	Off	On	On
On	Off	Off	Off
Off	On	On	Off
Off	On	Off	Off
Off	Off	On	On
Off	Off	Off	Off

If all this still sounds complicated, it really isn't. To implement it, all you need to do is, for each dot, (1) disp = disp OR mask, then (2) disp = disp AND (NOT icon) — assuming we're drawing white on black. And, really, you don't even have to do that, since the ToolBox will do it for you — most of it, anyway.

Icon/mask pairs are stored in 'ICN#' type resources ('ICN#' resources can actually store an arbitrary number of 32x32 dot patterns, but they most often store only two, an icon and its mask). It's these type of resources that the Finder uses to store icons as seen on the desk-top, and this is also the reason you can place one icon on top of another and get the results you would expect intuitively.

3. Using Icons Within an Application

Icons can be used by an application either to implement user interface icons, or simply to draw small graphical areas — for instance, a graphic display could be designed as a picture but small changes or additions made, depending upon the run-time context, with icon overlays.

In order to use an icon from within an application, the icon must first be designed and included in the application's resource file. REdit includes excellent facilities for designing icons.

Second, the icon must be read into memory from the resource file:

```
Handle iconHand;

    .
    .                   Initialize, especially QuickDraw.
    .
```

```
iconHand = GetIcon(Icon Resource # );
if (iconHand == NIL) Resource not found error ;
        .
        .
        .
```

Following this call, the icon is in memory and can be used in calls to other ToolBox procedures. (Note: GetIcon() actually calls GetResource('ICON',*Icon Resource #*).)

The simplest of the routines that can use the icon is PlotIcon(), which plots the icons within a given rectangle, as follows:

```
Handle iconHand;
Rect iconRect;

iconHand = GetIcon(Icon Resource # );
if (iconHand == NIL) Resource not found error ;
SetRect(&iconRect,left,top,right,bottom );
PlotIcon(&iconRect,iconHand);
```

This erases anything underneath the rectangle specified and replaces it with the icon. It will even "magnify" the icon if the rectangle is larger (or smaller) than 32x32.

PlotIcon() actually calls the QuickDraw routine CopyBits(), and in order to do anything more sophisticated with icons, we need to understand this routine. Simply put, CopyBits() takes one rectangular area and copies it onto another, combining the two together in one of several selectable ways. CopyBits() is a very fast, very versatile routine, which can be used in a large number of areas of drawing.

A really complete understanding of CopyBits() requires an understanding of many of the concepts behind QuickDraw, including bit maps, regions, and copy modes. All of which is beyond the scope of this article. Luckily, the QuickDraw section is perhaps the best written part of Inside Macintosh, and a thorough reading of that section should allow the reader to understand CopyBits() and how to use it. The remainder of this section assumes that the reader has acquired that understanding.

In order to use CopyBits() on icons, we must construct a bit map of the right size and shape for icons:

```
BitMap iconBits;

iconBits.rowBytes = 4;                          Set # of bytes per
row.
iconBits.bounds.top = iconBits.bounds.left = 0;     Set a 32x32
rectangle.
iconBits.bounds.bottom = iconBits.bounds.right = 32;
```

When we want to copy an icon onto the display, we must point the bit map's base address to the specific icon we are using. Then we can call CopyBits() just as we called PlotIcon() above:

```
Handle iconHand;
Rect iconRect;
BitMap iconBits;

iconBits.rowBytes = 4;                          Set # of bytes
per row.
iconBits.bounds.top = iconBits.bounds.left = 0;  Set a 32x32
rectangle.
iconBits.bounds.bottom = iconBits.bounds.right = 32;
            .
            .
            .
iconHand = GetIcon(Icon Resource # );           Get the icon.
if (iconHand == NIL) Resource not found error ;
            .
            .
            .
SetRect(&iconRect,left,top,right,bottom );      Set the
rectangle.
iconBits.baseAddr = *iconHand;                  Set the base
address.
CopyBits(&iconBits,&thePort->portBits,          Display it.
            &iconBits.bounds,&iconRect,
            srcCopy,NIL);
```

(Note: The base address assignment dereferences the iconHand handle. This must be done each time immediately before CopyBits() since memory management activity may cause the memory location of the icon to change.)

As long as the current grafPort's portBits is used as the destination bit map — as it is above — CopyBits() will clip the copy operation to the intersection of the grafPort's clipRgn and visRgn. This is as desired for displaying icons within windows, updating windows, etc.

Now that we've constructed the equivalent of PlotIcon(), we're in a position to start making extensions. Two extensions are needed in order to implement the masked icons described above: (1) we must allow copy modes other than srcCopy; and (2) we must allow access to 'ICN#' type resources. In order to do this, we'll define a function, called PlotIconPlus(), which has two parameters in addition to those of PlotIcon(): the first is the copy mode, and the second is the number of the icon in the resource, starting at zero.

```
PlotIconPlus(iconRect,iconHand,iconMode,iconNum)
```

```
Rect iconRect;
Handle iconHand;
int iconMode;
int iconNum;

{
        BitMap  iconBits;          /* The icon bit map for. */

        /* Set up the bit map. */
        iconBits.rowBytes = 4;
        iconBits.bounds.top = iconBits.bounds.left = 0;
        iconBits.bounds.bottom = iconBits.bounds.right = 32;
        iconBits.baseAddr = *iconHand + (iconNum*128);

        /* Copy in the icon. */
        CopyBits(&iconBits,&thePort-
>portBits,&iconBits.bounds,&iconRect,
                        iconMode,NIL);
}
```

It's now quite simple to use icons with masks, just like the Finder. First, we call GetResource() to read them into memory (we could define a new version of GetIcon() for use with 'ICN#' resources as well — this is left as a (rather simple) exercise for the reader). Then we clear the area under the mask. And finally, we draw the icon.

```
Handle iconHand;
Rect iconRect;

iconHand = GetResource('ICN#',Icon Resource # );      Get the ICN#
resource.
if (iconHand == NIL) Resource not found error.

SetRect(&iconRect,left,top,right,bottom );            Set the
rectangle.

PlotIconPlus(&iconRect,iconHand,notSrcBic,1);         Clear the mask
area.
PlotIconPlus(&iconRect,iconHand,srcOr,0);             Draw the icon.
```

Now we can easily define a routine which plots icons with masks:

```
PlotMaskIcon(iconRect,iconHand)

Rect iconRect;
Handle iconHand;

{
```

```
        /* Clear out the mask area. */
        PlotIconPlus(&iconRect,iconHand,notSrcBic,1);

        /* Draw the icon. */
        PlotIconPlus(&iconRect,iconHand,srcOr,0);
}
```

Most operations with icons can be accomplished using PlotIcon() and PlotMaskIcon(). PlotIconPlus() adds a large additional set of possibilities. Reverting all the way to CopyBits() allows the programmer complete freedom of action.

4. Finder Icons

(Note: This section assumes a knowledge of file types and creators, which the reader should acquire from Inside Macintosh. In brief: Each file has a type, which is the type and format of its contents ('TEXT' for example). Each file also has a creator (or signature), which indicates which program created it and also which program to invoke if the user double-clicks the document ('MPNT' for MacPaint for example). An application file has a file type of 'APPL' and itself as creator.)

The application programmer can tell the Finder which icon to use to represent the application on the desk top, as well as icons to represent other file types used by the application. This is done using 'FREF' type resources, which tell, for each file type, which icon/mask to use. File type 'APPL' refers to the application itself.

Sounds simple? Well, of course, it couldn't be *that* simple. So, the number given in the 'FREF' resource isn't the true resource number of the 'ICN#' resource. Instead, it's a "local" resource number, which is mapped into the true resource number in a 'BNDL' (bundle) type resource. The 'BNDL' resource must also include the application's creator signature. And the resource file must also contain a special resource called its autograph.

Still sounds simple? No, it doesn't to me either. An example should make things clearer: Suppose we create an application which will use one special format data file. If we use a creator of 'MYAP' and a data file type of 'MYDA', and if we have two previously created 'ICN#' resources, number 256 for the application icon/mask, and 257 for the data file icon/mask, the following RMaker source file will do the trick:

```
TYPE MYAP=STR                   The autograph.
,0
This can be any string you like, usually a version number.

TYPE FREF                       Application icon/mask is local ID 0.
,256
APPL 0
```

```
TYPE FREF                    Data file icon/mask is local ID 1.
,257
MYDA 1

TYPE BNDL                    The Bundle.
,256
MYAP 0                       The type and ID of the autograph.
FREF                         FREF mappings.
0 256 1 257                  Local ID 0 -> 256, 1 -> 257.
ICN#                         ICN# mappings.
0 256 1 257                  Local ID 0 -> 256, 1 -> 257.
```

Additional notes: In addition to these resources, there is also a "bundle bit" in the application file header which must be set in order to inform the Finder that this file has a bundle, etc. This bit is generally set by the development system you're using, but it may not be, or it may become reset if you move a file from one system to another under certain circumstances. Therefore, if you're sure you've done everything right, but the icons don't appear on the desk top, try using a program like FEdit to make sure that the application's "bundle bit" is set.

The creators and types must be unique, and unless you plan only to use the program yourself or in very limited circles, you should get these assigned from Apple Technical Support.

5. Summary

The article has described the use of icons in the Macintosh system. This includes their use from within applications as well as their use to represent those same applications and the associated data files in the Finder.

Cursors are fairly similar to icons, but are 16x16 dots rather than 32x32. As an exercise, the reader is encouraged to learn how to use cursors, both as controlled from the mouse and in ways similar to icons — for instance, as displayed in a palette. This information can be found in the Inside Macintosh sections on QuickDraw and the ToolBox Utilities.

Outside Outside Macintosh

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

◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊

How Big Is My Market

By Hazel Holby, *Outside Macintosh* Editor

One of the questions people frequently ask the Macintosh Developers Group is, "What vertical markets are recognized and supported by Apple?"

Answering this question isn't always easy, because we encourage and support development for all vertical markets. Our advertising and co-marketing dollars, however, are generally spent on generic productivity software products.

In order for us to understand the Macintosh niche in vertical markets, it is important to define these markets and their attributes. A recent market study — compiled by Kathy Jordan at Apple — lets us take a close look at some specific vertical markets.

In the study, Jordan states, "The small and medium-size business market can be thought of as a series of vertical markets. Market opportunities are tremendous — a total of 29.8 million white collar workers, comprising 67 percent of the total business market."

She selected eleven vertical markets for the study on the basis of the following criteria:

- Macintosh features and benefits
- Amount of influence over other potential customer groups
- Size of white collar work force
- Apple versus IBM share
- Penetration

She broke down the software requirements for vertical markets into three levels:

- Generic productivity (off-the-shelf horizontal productivity)
- Customized productivity (basic productivity that is tailored to meet vertical markets' needs)
- Vertical specificity (capabilities unique to the particular vertical market segment)

The accompanying tables present much of the data from this study.

Table I. Market Size and Penetration of Specific Vertical Markets

Market	Market Size (x 1,000 workers)	Penetration of Market (%)
Retail/Wholesale	5,433	7
Legal	4,121	25
Advertising	1,755	11
Consulting	1,478	10
Medical/Dental	1,249	7
Real Estate	1,018	10
Architecture/Construction	951	10
Agriculture	872	10
Food/Lodging	695	9
Accounting	418	18
Printing	225	10

Table II. Characteristics of Specific Vertical Markets

Market	Hardware Needs	Purchase Habits
Retail/Wholesale	• 256K RAM • 20MB hard disk • Point-of-sale tie-in • Bar-code-reading tie-in very	• Buys from salesperson calling at place of business • Shrewd buyers • Trade association endorsement influential
Legal	• 512K RAM • 10MB hard disk • Letter-quality printing	• Buys directly from VARS
Advertising	• 512K RAM • Letter-quality printing	• Buys from computer stores • Buying decisions made a high level • Purchases based on software needs

Market	Hardware Needs	Purchase Habits
Consulting visit	• Portable to transportable • 512K RAM • 20MB hard disk • Letter-quality printing	• Large firms (>10) want rep to location • Small firms (<10) will visit showrooms or computer dealers
Medical/Dental at	• 512K RAM • 10-20MB hard disk • Printer able to print variety of 3-part forms	• Buys from salespersons calling office • Long selling cycle (2 mo. min.)
Real Estate	• Networking • 512K RAM	• Buys from computer stores • Can also buy through National Association of Realtors
Architecture/ Construction	• 10-20MB hard disk	• Buys from retail outlets • Uses consulting services • Firm owner makes purch. decision
Agriculture	• 512K RAM • 20MB hard disk	• Tends to buy from agriculture supply stores • Credibility, knowledge of market is important
Food/Lodging	• 256-512K RAM • 10MB hard disk	• Buys at trade shows • Uses direct mail (advertising in trade journals) • Restaurants might visit a retail store hotels prefer salespeople
Accounting	• Hard disk	• Buys from computer dealers and office equipment dealers • Practical, cautious purchasers
Printing/ Typesetting and them	• 600-800 DPI printer would substitute for offset printing	• Large printer: - Equipment vendors call on - Trade association influential • Small and medium-size printers "quick printers": - Sales call or telephone call

speak-

ing to new technology — will

go

to demo
- Trade assoc. recommendation
- Will go to computer store for generic software

Table III. Strategic Market Software Requirement

	Real Estate	Professional Accountants	Agriculture	Food/ Lodging	Retail/ Wholesale
Generic Productivity:					
Word Processing	•	•	•	•	•
Database	•	•	•	•	•
Spreadsheet	•	•	•	•	•
Calendaring	•	•		•	
Cash Flow	•	•	•	•	•
Graphics	•				
Project Management	•				
Customized Productivity:					
Accounting	•	•	•	•	•
Sale Force Mgmt.	•				•
Client/Patient Billing		•			
Client/Patient Records					
Invoicing			•	•	•
Inventory Mgmt.			•	•	•
Bar Code					•
Vertical Applications:	Load Qual. Multi-entry Listing	Auditing Tax Prep.	Commodity Charting Cow/calf Mgmt.	Point of Sale Menu Planning	Point of Sale Order

	Medical/ Printing Dental	Advertising	Legal	Arch./ Const.	Consulting
Generic Productivity:					
Word Processing	•	•	•	•	•
Database	•	•	•	•	•
Spreadsheet	•	•		•	•
Calendaring		•	•	•	•
Cash Flow	•	•		•	•
Graphics	•	•		•	•
Project Management	•			•	•
Customized Productivity:					
Accounting	•	•	•	•	•
Sale Force Mgmt.		•			
Client/Patient Billing		•	•	•	•
Client/Patient Records		•	•		•
Invoicing	•				
Inventory Mgmt.	•			•	
Bar Code	•				
Vertical Applications:	Estimating Pharmacy Quotations Mgmt. Typesetting	Legal	Estate Taxes Estimates Dict.	CAD Cost	Report Composition

Source: *Future Computing (primary), Microcomputer Research Group, VARS Study, American Association of Advertising Agencies, National Association of Realtors, National Cattleman's Association, American Institute of Certified Public Accountants, Printing Industry of America.*

◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊

NEON Development System Uses Full Power of Macintosh

By Reese Warner, Kriya Systems, Inc.

Though the design of the Macintosh was influenced by the machines designed at Xerox PARC, the exciting programming style developed for those machines has not yet been made available for the Macintosh. NEON, however, is an object-oriented programming language for the Macintosh that allows developers to use the powerful Smalltalk programming concepts to create developer-quality applications.

The full power of the Macintosh is available with NEON: any Toolbox routine can be called from within a NEON application. But more than this, NEON provides a higher-level Toolbox interface using the class mechanism, so you don't need to know *Inside Macintosh* to create exciting software.

What's on the Disk?

Included on the disk in source form are classes that support windows, menus, controls, strings, ports, drivers, events, files, dialogs, and QuickDraw objects (such as points, rectangles, graphics, pictures, and images).

For faster development, several programming utilities are integrated with the NEON environment at the click of the mouse — utilities such as a memory-examiner window and a grep-like (pattern matching) file-search utility.

The Install utility creates double-clickable applications that seal off the interpreter from your end-user. Such sealed applications can be distributed with no further licensing fee due to Kriya.

Due to its FORTH ancestry, NEON provides the advantages of the interpreted languages — extensibility and elimination of the Edit-Link-Compile-Run cycle of development — while being efficient both in terms of space and time. NEON is a highly extended version of FORTH, 83-Standard, so FORTH programmers will feel at home.

But whether new to FORTH or more experienced, programmers won't be limited to FORTH programming constructs. NEON provides for local variables within word definitions, named input parameters, and richer data structures built within the class-object framework. Classes supporting arrays and ordered collections come with the system.

Is There a Manual?

The system also comes with a 500-page manual containing both tutorial and reference sections. The tutorial, written by Danny Goodman, a contributing editor for *MacWorld* magazine, shows step-by-step how to program in NEON. The reference section, written by our programming staff, complements the tutorial.

In addition, a complete demonstration application comes with the system. This application, a graphics program, uses controls and menus. The manual's tutorial section examines the source code in depth, and includes instructions that explain how to make the demo into a double-clickable application.

The NEON system comes with a window-based editor as well, implemented as a desk accessory that allows multiple windows on an XL or a 512K machine. NEON and its editor use files in the standard Macintosh ASCII format.

Advantages of the Module System

The NEON module system allows dynamic overlays: sections of relocatable code that reside on disk until they are necessary. Modules are loaded on the heap and remain until the space they occupy is needed for another purpose. It is also possible to dynamically create objects and keep them on the heap.

Most of NEON's programming utilities are implemented as modules. Though it's possible to develop applications on a Macintosh 128K, developing on a 512K allows faster use of the programming utilities, and is recommended. However, Typing Tutor III with Letter Invaders, the best-selling educational program, was developed in NEON on a 128K Macintosh.

NEON is a growing language. Classes to support new features will be available soon, including floating point, linked lists, symbol tables, and a general purpose quick-sort routine. An integrated 68000 assembler will also be available shortly as an add-on package.

In summary, NEON is a full development language, providing object-oriented programming in a language fast enough to write usable commercial software. Applications can be created quickly using a powerful development environment, and sold with no concern for licensing fee. NEON is available now for $155 from Kriya Systems.