

Frequently Asked MPW® C++ and MacApp® questions:

Version: 1.0a4

Date: 08/22/91

Moderator: Kent Sandvik, Apple® Developer Technical Support

Concerning changes, comments, updates, corrections, right to use and any other correspondence, contact:
AppleLink: KSAND

Usenet: ksand@apple.com

CompuServe: 75300,1331

For more updated information, consult the Developer Services Q&A stack, available on ftp.apple.com (Internet), and on the Developer Services CDs.

FREQUENTLY ASKED C++ QUESTIONS:

1. How do I use MPW C++ load/dump, how do I use it with MacApp?

A) To start with, make sure you have the right C compiler for dump/load. You need to have the C 3.2.31 compiler included with the MPW C++ 1.0 final, or a newer version, like MPW C 3.2.

To test this, execute the following instructions from the MPW shell:

```
CPlus "{CIncludes}"stdio.h -dump stdio.dump
```

If the C compiler complains about "unexpected tokens" you have the wrong C compiler version.

B) If this worked, the next step is to get it working with a generic C++ program. The example given in the C++ release notes (C++ 1.0 final) is a good one that should work. Use the rules in the release notes to build a single uniform header file that will be used to include the other header files.

C) Let's now try with MacApp. To start with, MacApp 2.0 has a new flag for MABuild which takes care of the whole dump/load management concerning the MacApp C++ header files. If you include a -CPlusLoad to MABuild the load/dump will trigger automatically with the basic MacApp header files dumped and loaded. The dumped file is stored into a folder under the MPW folder called "Load Files." In the {MacApp} folder there's a file called Startup where you can modify where the dump file is stored; you could also specify that the -CPlusLoad is done automatically each time.

In this case the dump/load management is handled for you. The drawback is that you are not able to include your own header files.

D.1) How to include your own header files: There are two different ways to do it. The first one is to modify a file in the {MacApp}Tools folder called 'Build Rules and Dependencies', and include the files that will be loaded and dumped. If you search this one you will find a place where the dump/load is defined. Comment out the line that says, "Include you own files here.", and insert the additional load/dump files.

D.2) Now let us assume that you want to build your own MAMake file. The trick in this case is to carefully specify where the dump, source and object files reside by using the prefix {ObjApp} and {SrcApp} for each file. The other trick is to have a dummy object file as one of the dependency rules that builds the dump file in the first place. It is important that the OtherLinkFiles variable has this dummy file in its definition.

The dump/load is about 2-3 times faster during the C compilation phase due to the fast load of precompiled information. However, if you modify the dumped files MPW needs to generate a new dump file, which takes

some additional time. An example of an MAMake file for MacApp ("DemoText with C++ load/dump") is on AppleLink, on the Developer Services CD, or available upon request from DTS.

2. I can't compile any of the MacApp C++ example programs without getting a CFront "free store exhausted" error from CFront. I cannot find any reference to this diagnostic in my MPW documentation. The example C programs compile OK.

Because the MPW C++ compiler is a port of the AT&T CFront, which was designed with the assumption that all computers in the world have virtual memory paging (spell UNIX®), it makes notorious use of memory. It doesn't care about the free memory because it assumes that unused memory will be paged out temporarily and new free pages will be available for the compiler. Enter PC and Macintosh® systems, with no virtual memory paging. This is the reason the MPW C++ needs a lot of temporary memory.

Fortunately the peak "need memory" points are infrequent, mostly during the parsing phase. A switch called "-mf" can be used with all the MPW compiler, lib and link tools. It uses the MultiFinder® temporary memory, which is not in use normally and is sort of a pool of unused memory.

If you include the -mf switch with the set of Cfront options, either at the MPW Shell level, in the Makefile, with MAMake using the -Cfront -mf, or even defining it in the Startup file inside the MacApp folder as one of the default Cfront parameters, then you should have a far easier time compiling MacApp/C++ code.

If this does not help, increase the MPW Shell application heap size as well (Get Info on the MPW Shell icon in the Finder™). The shell size should be at least 2.5 MB, or up to 3 or 4 MB if possible. If you have even more memory and you use MPW most of the time, increase the size to near the top of the memory range. Sometimes you also need to increase the stack size (HEXA 128), instructions about how to do this are to be found in the MPW C++ release notes.

3: Where do I find information about how the vtables are generated, we have noticed that the virtual table sizes in the output file are different for different objects, and we would like to know how these are generated?

The topic concerning generation of vtables is very wide. Apple's MPW 3.1 C++ compiler originates from AT&T Cfront 2.0, and it generate vtables in the same manner as the original C++ compiler.

You can find information about virtual tables and how the tables are generated/accessed from the following sources:

"MPW C++ 3.1 Release Notes"

Macintosh Technical Note #281, "Multiple Inheritance and Handle Objects"

"Multiple Inheritance for C++," included with MPW C++
Chapter 10 of "Annotated C++ Reference Guide", by Ellis and Stroustrup

4. I can't get pure virtual functions compiled with C++ classes that have PascalObject as the base class.

Due to a limitation with the MPW linker pure virtual functions are supported only if the virtual function is defined with a dummy stub. As an example the following pure virtual function definition works:

```
class TAbstract : public PascalObject{
    virtual void Method() =0;           // defined abstract
};

void TAbstract::Method()
{ // we need to create this dummy member function for the linker
}

class TDerived : public TAbstract{
    virtual void Method() {};          // now finally defined
};

TAbstract* noWay;
TDerived* OK;

main()
{
    noWay = new TAbstract; // the compiler will complain!
    OK    = new TDerived;  // no problems.
```

For a more complete record of known limitations with PascalObjects and HandleObjects, consult Tech Notes 265, 281 and 300.

5. I defined a couple of member functions to be inline, and C++ did redefine them to be outlined, and caused me problems with the segmentation. Is there any way to know when this happens? Why can't I force the compiler to always inline my inline C++ statements?

Inline statements are hints to the compiler, which could ignore the hint based on a variety of issues, like:

- a) if the call is recursive
- b) if the call returns values based on the result of an input operation
- c) if an address to the inline function is taken
- d) if a call to an inlined function through a pointer to a function
- e) if an inline function was too large for inlining to be worthwhile
- f) if an inline function was invoked in a program before it was defined
- g) if an inline function was invoked twice within an expression
- h) if an inline function contained a loop, a switch, or a goto

These rules are highly implementation dependent, and the rules listed above are based on the rules for AT&T CFront.

There is a switch for making all inline calls to function, which is recommended to use if something is not

working properly, and this flag is the:

```
-z0          # force 'inline' functions to be non-inline
```

The book "Annotated C++ Reference, page 102, has a good design description why inlining is a hint and not a solid operator. Also another drawback is that inlining has to be done in the header files, which opens the design for the rest of the world.

Starting with MPW 3.2 C++ the -w flag will warn when an inlined function is not inlined by the compiler.

5. When I use SADE® with C++ code I have noticed `__Wddd` fields inside the classes which takes 8 bytes each. Why are these generated, are they unnecessary?

The `__Wddd` are placeholders for addresses in Cfront (AT&T), so classes that do not contain any members could be found (addressing). For instance these are generated in the case of empty base classes. If the base classes contain at least one field the `__Wddd` is not generated.

6. Until C++ has failure handling, how could I implement something similar?

a) Check out the DTS Sample code example number 14, where the MacApp failure handling code is used for C++ macros, which implements the a failure handling syntax. Tech Note 88 talks more about the signal error trapping system recommended by DTS.

The C++ language does not have the volatile keyword implemented, which would help in order to define that certain variables should never be register allocated (concerning error handling).

The following macro will do the same job:

```
#define VOLATILE(a) ((void) &a)
```

7. I have problems with overloading the new operator with HandleObject base objects, the compiler returns "error: bad return type for operator new()" for my overloaded new function.

The reason you get the errors about wrong type when you are overloading the new operator - which is part of the HandleObject - is that the HandleObject has already overloaded the new operator. And instead of returning (void *) like the normal new operators, it returns a handle, or (void **). So if you change your code so it will return handles, then the operator new overload should compile.

8. I need to call a C++ routine from within a C program. Is there an elegant way for me to do it? I wish that I could just run the C code thought the C++ compiler, but unfortunately that code must also run through a compiler that doesn't use function prototypes.

A simple workaround with the function prototypes is to ifdef two different function headers, like:

```
#ifdef __cplusplus int foo(int bar);
#else
    foo(bar);
```

If you have the C++ function already compiled and is part of an object library, then you could use the

mangled function name, or specify extern "C" for the function inside the C++ sources, so the C++ compiler would not mangle the name when it builds the object code.

9. I get link errors "Undefined entry, name: (Error 28 "__ptbl__4TFoo", even if I know the C++ code looks OK. What could cause this linker error"

Most likely you have missing virtual tables, because evidently you defined a virtual function in TFoo, but you did not implement this member function.

10. I tried to define enums and use these as parts of member function prototypes, however the compiler has problems with these member functions. What is wrong?

Evidently you assigned the enums after the member function statements, as in:

```
class TFoo{
public:
    theResult Foo(void);
....
    enum theResult(err = 0, maybe = 1, OK =3);
};
```

and Cfront is a one-pass translator so it does not know about the enum until it has parsed the enum statement.

--

11. How is MPW C++ different from AT&T Cfront?

MPW C++ 3.1 is based on AT&T Cfront 2.0, it contains additional bug fixes. MPW C++ 3.2 will be based on the Cfront 2.1 release.

MPW C++ do have additional functionality, mostly biased towards Macintosh toolbox programming. It does support Object Pascal method dispatching, which makes it possible to link together Object Pascal code and C++ code (as with MacApp).

MPW C++ has support for pascal based function calls (pushing arguments to the stack from left to right), support for 68881 code generation, SADE libraries, Pascal string support, Macintosh memory model support (HandleObjects). MPW C++ also has a new keyword called inherited, which calls inherited member function (this only works with PascalObject based classes, as in MacApp).

12. Where do I get more information about MPW C++?

Please consult the MPW C++ documentation (MPW C++ Manual, MPW C++ release notes). Tech Note 265 describes how to convert Pascal nested procedures to C++, Tech Note 281 talks about why Multiple inheritance does not work with HandleObject based C++ classes, Tech Note 300 maps the known problems and bugs with PascalObject based objects.

AppleLink has a forum for developers concerning MPW C++, also there's a mailing list on AppleLink called CPLUS.DEV\$ where developers discuss about MPW C++. Old entries from this mailing list is available on

Finally, "Elements of C++ Macintosh Programming" by Dan Weston is a good book about MPW C++ and Macintosh programming.

Frequently Asked MacApp Questions

1. What is the recommended method of segmentation for MacApp applications? If I pull out my object modules and test them by themselves they function correctly. My application has now grown beyond my previous segmentation scheme and I'm back to the drawing board.

Yes, it makes sense to plan for segmentation with MacApp, mostly because the code segments tend to become very crowded, and also because Macintosh operating system does not have virtual-page memory paging. So it is the responsibility of the programmer to define where certain functions are in various segments. And MacApp will load those segments it needs — so the developer is in a way defining dynamically loaded libraries (and it makes sense to push certain methods which are seldom called into segments that reside most of the time on the hard disk instead of in the memory).

So try to separate all the methods in more segments, and to group methods that are related to the same segments. If possible create small segments, which load faster from disk to memory.

Here's a suggestion list of where to place various methods in what segments (taken from the Dave Wilson MacApp training manual):

Segment name - routines

ARes	- routines that get called often DoSetupMenus DoSetCursor DIdle Draw DoKeyCommands, if typing
ADebug	- debugging code
AFields	- your Fields methods
Alnit	- used only once (during the lifetime of the application) IYourApplication
ATerminate	- used only when you quit

ASelCommand - for selecting commands

- DoMenuCommand
- DoMouseCommand
- DoKeyCommand
- TYourPasteCommand
- IYourCommand

ADoCommand - for performing commands

- TReColorCmd::Dolt
- TSketcher::TrackMouse
- TTypingCmd::Commit

AClipboard - for clipboard non-commands

- MakeViewForAlienClipboard
- GivePasteData
- WriteToDeskScrap

AOpen - opening stuff

- DoMakeViews
- DoMakeWindows
- IYourDocument
- DoMakeDocument
- IYourView

AClose - closing stuff

AReadFile - reading from disk

- DoRead
- DoInitialState

AWriteFile - saving files

- DoNeedDiskSpace
- DoWrite

AFile - rarely used

ANonRes - for routines you rarely call

ASomething - define your own segments and place related methods here

Hint: If you are unsure where a method goes, put it in same code segment as the method that calls it. The Mouser browser has a nice feature that shows the segment in which each method is placed.

Another issue is that place segments that you don't want to be unloaded automatically from memory in the 'res!' resource (good for methods that are time critical, and should not be unloaded to disk), and use the 'mem!' resource for defining how much to add to the basic temporary memory reserve memory size, permanent memory reserve, and stack space.

For more information about MacApp segmentation, read the article "Segments from Outer Space" in the June 1991 issue of MADA FrameWorks magazine.

2. My application crashes on Macintosh SE but not Macintosh II systems while attempting to read a block of code into memory and parse it. The routine passes a pointer created by NewPermPtr to various objects and so that they can extract information and increment the pointer. The MacApp debugger gives me the following message:

```
Exception #3 Address error: Word or long-word reference made to an odd
address
Bad address was: INVALID! ($0005372D)
```

You can get odd-address errors with 68000-based machines. Starting with 68020 the CPU is able to fetch data from odd addresses. It's a common problem to get into trouble with data fetched from odd addresses with 68000-based platforms. Possibly the data you are saving and later retrieving has odd length, causing the next fetch to start on an odd address.

To guarantee that the data will always start on even boundaries, use the MacApp OffsetPtr global routine. Here's the new MacApp 3.0 C++ version of this routine:

```
pascal void OffsetPtr(Ptr& p, long offset)
{
    p += offset;
    if (((long) p) & 1)
        ++p;
}
```

The function tests if the pointer is pointing at an odd address, and if so it increases the pointer one byte. You could do something similar in your member function, such as:

```
short* sptr = (short*)buf;
fValue = *sptr;
OffsetPtr(buf, sizeof(short)); // instead of buf+= sizeof(short);
```

3. Where can I find information about writing a desk accessory (DA) in MacApp?

Take a look at Technical Note #239, "Inside Object Pascal", which explains why, as an application framework, MacApp should be used to write only applications and MPW tools. The framework itself does not support standalone code development.

4. Where/when is the best place to extract text from my fill-in-the-blank Macintosh dialog with several TEditText views? Does a method get called when another field is selected or another view (like a button) is chosen? I can think of several ways to accomplish what I want to do, but I'd rather use the MacApp view architecture the way it is DESIGNED to be used.

You can get one answer to this by looking in the DemoDialogs example (C++ or OP). TView has a method called DoChoice, which should be activated, for instance, when a button, check box, or even with TrackMouse operating over a particular view or groups of view, has triggered something, and the application wants something to be done. Using MacBrowse™, do a “Find References” (or Command-R) to find all references wherefrom DoChoice is called, such as TrackMouse or DoKeyCommand. Fields are Views, so they will inherit DoChoice, and in the case of TEditText you could, for instance, call DoChoice from HandleMouseDown (when clicking in a particular field).

HandleMouseDown is also a TView method, so most views have this one implemented. Special sub-Views have additional indicators that something was triggered inside the view.

5. What is the C++ equivalent for the {\$D±} Object Pascal compiler directive? I'd like the same debugging options for C++ that exist for Object Pascal.

The MacApp debugger is supported with C++ Object Pascal object information. In other words, you can't trace other objects than those derived from PascalObjects.

This works OK if you include a “-trace on” or “-trace always” with CFront when you compile your MacApp source code. We assume that you are using the latest MPW C++ (3.1 final and MPW 3.1 C tools). MPW 3.2 C/C++ also supports a pragma called #pragma trace on/off, which emulates the {\$D+-}.

In general the S (stack info) command with the MacApp debugger has a lower signal/noise ratio than the R (recent PC) command concerning useful information, but both work OK.

6. Is there a way to add floating windows to a MacApp application?

MacApp 2.0.1 contains experimental-and-unsupported code, and the floating window package is part of this code. Please check the source code (MacApp), and use the flag qExperimentalAndUnsupported for the build of the floating window code.

Official support for floating windows will be available from MacApp 3.0 forward. Check the DrawShapes MacApp 3.0 sample for code showing how to program floating windows.

7. Do objects behave like handles in the sense that they can and should be locked when passing a field as a VAR parameter? If so is the solution the same as with handles - HLock and UnLock?

PascalObjects are handles, so in general they tend to move in memory if there is a memory compaction activity going on. A good example that might happen in the MacApp world is that you call a certain function with the field of a class, and the segment where the function is residing in is not loaded into memory. So the

Segment Loader loads it in, and at the same time tries to compact the memory. So we end up with a pointer that points at nowhere.

If you call the field from the same object, and the method is in the same segment as the other methods used earlier, and you are sure there is no trap or activity that might move memory, then it is OK to make use of the field. This is quite typical in most cases, because eventually the object is a global one, in which it resides in the A5 world which does not move, or you use the field to assign data to other fields that exists in the same handle.

In general TObjLock and TObjUnlock are a little bit costly, so we recommend that you use a temporary variable that you fill with the field value, and pass this one to the other function (which is much faster). For more good information, read the "Using Objects Safely in Object Pascal" article in d e v e l o p issue 2 1990. The article should be on any of the Developer Services CDs.

8. If an "IncludeViewsAt" directive does not immediately follow the parent view, running the MacApp program and attempting to display the view produces a "can't find parent view" message from the MacApp debugger. Is this requirement documented somewhere? Why is this required?

The TEvtHandler.DoCreateViews doesn't work right if you build your view tree in the 'wrong' order, i.e. breath-first order. If you declare them as ViewEdit/DeRez originally has defined them, as a hierarchy of levels, then the problem will go away.

9. I have problems with the compiling and linking times with MacApp and MPW. Are there any known tricks to speed up the development work.

Here are some tricks to improve the overall development time with MacApp:

- a) Use Think Pascal (works with MacApp 2.0(.1).
- b) Use Steve Jasik's IBS (Incremental Build Linking utility for MPW)
- c) Use a RAM disk, offload library files, scratch files and possible C++ load/dump files to this location
- d) Use a bigger MPW heap size partition
- e) Make use of C++ load/dump with MacApp (MABuild -CPlusLoad flag)
- f) Use faster Macintosh hardware
- g) Use SCSI-2 cards and other File I/O improvement tricks (faster hard disks)
- h) Increase the System cache to 256k.
- i) Don't have Personal FileShare, and other networking Inits running on the development machine if possible
- j) Try to include as few header files as possible, use the C++ #ifdef __FOO__ trick (or #pragma once in MPW 3.2)
- k) Try to compile as small files as possible, split the contents of the files to smaller units, and build MAMake files that take care of the dependencies.

10. I have problems with MPW 3.2 model far support and MacApp, I can't get it to work.

Model far support in MacApp 2.0(.1) is unsupported, there are files on the ETO disks which provide model far support, but these patches are unsupported and experimental. The main issue is to install all the needed

files, and build the new MABuild tool. Then you should use the -MedalFarCode (more jump table entries) or -ModelFarData (bigger global space) flags with MABuild.

Starting with MacApp 3.0 model far support is officially supported.

11. I have a problem with my MacApp code, when I launch my application I get an Alert box that says "Could not create a new document because of a program error". I have traced the problem to a View, which I load dynamically and then issue a Draw call, in which the program stops? What is happening, I could compile the application without problems?

Most likely the linker has stripped out the object, because the linker does not know that an MacApp object could be created dynamically (using a ViewServer or a WindowServer in MacApp 3.0). The way to help the linker is to state that a certain class should not be stripped out. This is usually done in the IApplication method:

```
if (gDeadStripSuppression)                // C++, MacApp 2.0(.1)
{
    TMyView *aMyView;
    aMyView = new TMyView;
}
```

```
if (gDeadStripSuppression){                // C++, MacApp 3.0
    macroDontDeadStrip(TMyView);
}
```

```
IF gDeadStripSuppression THEN              { Object Pascal, MacApp 2.0(1.) & 3.0
    BEGIN
        IF Member(TObject(NIL), TMyView) THEN;
        END;
    END;
```

Always doublecheck that classes not created by NEW (new in C++) have been declared non-strippable for the linker.

12. How do I print borders and page numbers with MacApp?

a) Override your TStdPrintHandler AdornPage() method (C++ example):

```
#pragma segment ANonRes
pascal void TMyPrintHandler::AdornPage()
{
    TextFont(appIFont);
    TextFace(normal);
    TextSize(12);
    this->PrintHeader();           // add these three methods to the class
    this->PrintPageNumber();
    this->PrintInkBorder();
}
```

b) Print Header:

```
pascal void TMyPrintHandler::PrintHeader()
{
    Point    headerLoc;

    headerLoc.h = fPageAreas.theInterior.left;
    headerLoc.v = fPageAreas.theInk.top + 20;    // 20 pixels down
    MoveTo(headerLoc.h, headerLoc.v);
    DrawString("\pHeader String");
}
```

c) Print Page Number:

```
pascal void TMyPrintHandler::PrintPageNumber()
{
    Str255    numberStr;
    Point    numberLoc;

    numberLoc.h = fPageAreas.theInterior.right - 30;
    numberLoc.v = fPageAreas.theInk.bottom - 10; // 10 pixels up
    MoveTo(numberLoc.h, numberLoc.v);
    NumToString(fFocusedPage, numberStr);
    DrawString(numberStr);
}
```

d) Print border:

```
pascal void TMyPrintHandler::PrintInkBorder()
{
    Rect inkRect = fPageAreas.theInk;
    PenNormal();
    FrameRect(&inkRect);
}
```

13. I have problems with MPW 3.2, MacApp 2.0(.1) in combination with heap fragmentation. Is there something that is causing all the fragmentation?

MacApp versions 3.0 a2 or earlier (including MacApp 2.01 and prior) have a bug when compiled with MPW 3.2. Due to some re-segmentation in the MPW Libraries, if you use a version of MacApp that has been compiled with 3.2 you may notice some serious heap fragmentation in your MacApp applications. We noticed this bug when calling SetHandleSize (it promptly failed while attempting to grow the handle). The primary solution to the problem is to use MPW 3.1 when compiling MacApp version 3.0 a2 or earlier.

If you REALLY need or want to use MPW 3.2, you must make the following modifications to your Basic Definitions (this workaround has not been thoroughly tested so... USE AT YOUR OWN RISK!):

```
SegmentMappings = 0
#---- insert here
-sn PASLIB=Main 0
```

```
-sn STDCLIB=Main 0  
#---- end of insertion
```

modify the following declaration in your MacApp.r (in 3.0 this declaration will be found in Memory.r)

```
resource 'seg!' (kBaseMacApp,  
#if qNames  
    "BaseMacApp",  
#endif  
    purgeable) {  
    { "Main";  
      "MAMain";  
      "GMain";  
      "GRes";  
      "GRes2";  
      "ARes";  
      "BBRes";  
      "BBRes2";  
      "%_MethTables";  
      "GError";  
#---- insert here  
      "INTENV";  
      "SADEV";  
      "INTENV";  
      "STDIO";  
      "PASLIB";  
      "STDIO";  
      "STDCLIB";  
#---- end of insertion
```

```
resource 'res!' (kBaseMacApp,  
#if qNames  
    "BaseMacApp",  
#endif  
    purgeable) {  
    { "Main";  
      "MAMain";  
      "GMain";  
      "GRes";  
      "GRes2";  
      "ARes";  
      "BBRes";  
      "BBRes2";  
      "%_MethTables";  
      "GError";  
#---- insert here  
      "INTENV";  
      "SADEV";  
      "INTENV";  
      "STDIO";  
      "PASLIB";
```

```
"STDIO";  
"STDCLIB";  
#---- end of insertion
```

One other possible solution would be to mark code resources produced by the libraries that were once in main as locked. Then, these segments would be loaded into memory and placed with the main segment, avoiding fragmentation problems. This can be done by modifying the user variable OtherLinkOptions in the Basic definitions file:

```
OtherLinkOptions = 0  
-raPASLIB=resLocked0  
-raSTDCLIB=resLocked
```

It would still be necessary to modify the SEG! resource as described above. Again, remember these fixes haven't been thoroughly tested, so use them at your own risk.

14. Where do I get more information about MacApp development?

MacDTS is one option, though there are also other avenues of tech support for MacApp programmers. The first is the group address MacApp.Tech\$. This is a group of MacApp developers on AppleLink that ask and answer questions of each other. You can simply send your questions to that address and request replies back to your personal account, or you can join the group and receive copies of all the mail sent to it by sending a request to MacApp.Admin.

Another avenue of support is from the MacApp Developer's Association. This is a non-profit organization set up by MacApp lovers. It's got about 1500 members across the nation, and even has a European counterpart. It has an annual MADA conference and a bimonthly magazine called FrameWorks. If you are interested, call them at (206) 252-6946, or write to them at AppleLink address MADA. Annual fee is \$75.

TradeMark information:

UNIX is a registered trademark of UNIX System Laboratories.

Apple, Apple Logo, SADE, Macintosh, MPW, MultiFinder and MacApp are registered trademarks of Apple Computer, Inc.

Finder and MacBrowse are trademarks of Apple Computer, Inc.