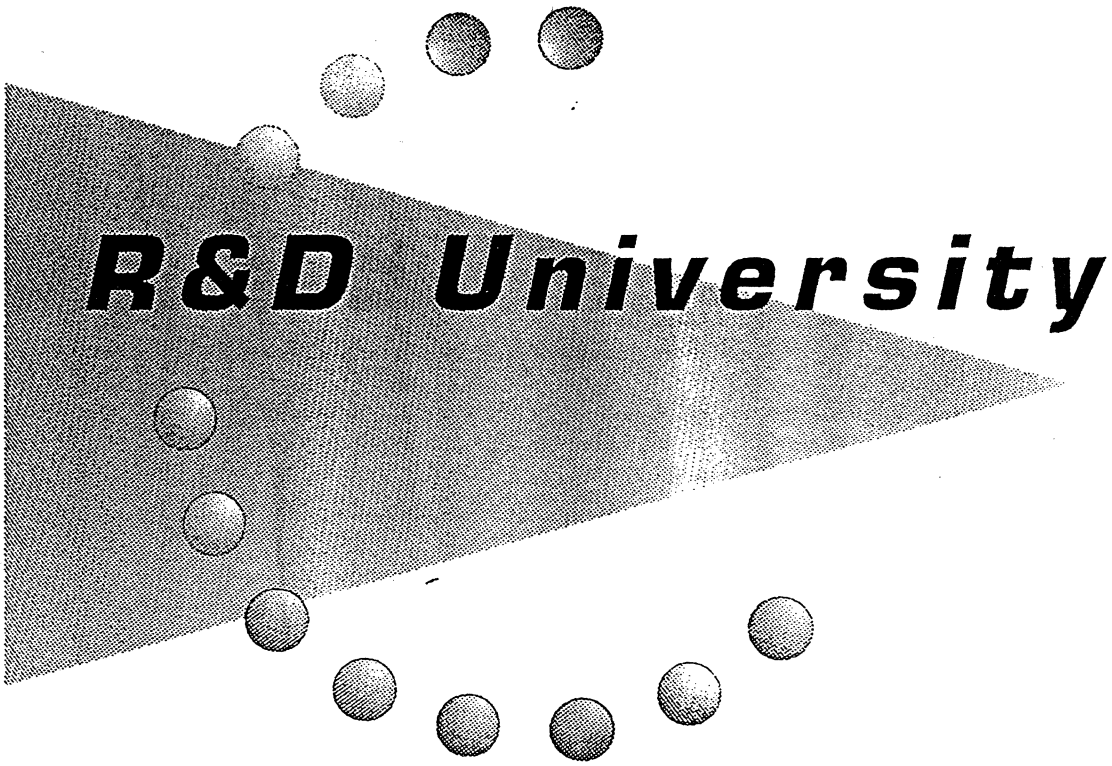
Porting to PowerPC



Migration strategies for PowerPC

C and C++ have become the languages of choice in this decade, and most of the development tools that are available are geared towards these languages. Because of this, the earliest PowerPC development tools will be oriented towards C and C++ programmers. However, since many developers have written their products in languages other than C or C++, we'll take a quick look at their options for moving to PowerPC.

Pascal Development

Pascal developers have a few options when moving their code to PowerPC:

- Wait for a third-party Pascal compiler: MetroWerks is producing a Pascal compilers for PowerPC. However, this compilers will not support Apple's "Object Pascal" extensions, so developers who were using Object Pascal (typically with MacApp) will need to use a different approach.

- Re-code in C or C++: This approach, while potentially difficult, gives you the widest range of portability options to PowerPC and other platforms (if you should ever choose to go cross-platform).

- Use a Pascal to C conversion tool: Sierra Software has produced "p2c", which converts Pascal or Object Pascal to C/C++. Several groups within Apple have used this tool, and most of them didn't like it. We're still looking for a better solution.

- Use a binary conversion tool: Echo Logic (a spin-off of Bell Labs) makes "FlashPort", a tool which treats your compiled application as the input to a compiler which emits PowerPC object code. This yields an application which is faster than a strictly emulated 680x0 application, but somewhat slower and larger than an application re-compiled for PowerPC from its original sources.

- Port as much as possible, and emulate non-portable parts via Mixed mode: This is a reasonable strategy if you have some parts of your code that are not processor intensive and which would be difficult to port. However, remember that you will not get the full performance benefits that a more complete porting job would yield.

Assembly-Language Development

Assembly-language programmers cannot simply re-compile for PowerPC; some sort of porting will be required.

Our recommendation is that you re-write your assembly code as portable C source code. While many assembly language programmers feel that this is a poor choice, the design of the PowerPC Macintosh removes many of the reasons that developers write in assembly language.

The 4 major reasons developers write in assembly language are 1) speed, 2) linking different calling conventions together, 3) addressing custom hardware installed in a particular machine, or 4) a need to jump to a particular routine instead of using a subroutine call. We'll look at how the PowerPC programming model addresses each of these issues.

- The first case, speed, should be handled adequately by the optimizing C compilers. Creating optimized code for the PowerPC is no easy feat, and it is easy to overlook one rule or another when hand-optimizing code. Additionally, a piece of code that is perfectly optimized on one version of PowerPC may be slightly sub-optimal on another implementation. The C compilers do an excellent job of optimizing your code, and you can re-compile for different PowerPC implementations much more easily than you could re-optimize your assembly code for each new processor.

- The second case, multiple calling conventions, has been eliminated by the uniform calling conventions for all PowerPC code, and Mixed Mode handles this for non-PowerPC code which calls PowerPC code (and vice versa.)

- Case 3, addressing custom hardware, is inherently non-portable, and should be done through the appropriate managers.

- This leaves the final case — jumping to a particular location. This typically occurs at the end of a "head patch" to a trap. The PowerPC Macintosh uses Mixed Mode to invoke traps, via the "CallRoutineDescriptor'" command. You should use this command to jump to a patched trap instead of using an explicit branch. (Note that since CallRoutineDescriptor will return control to your code after the patched trap executes, all patches become both head and tail patches.)

If you choose not to re-write your assembly language code as portable C, you have 2 other porting options: re-write into PowerPC assembler or use a conversion tool to convert your 680x0 code into PowerPC assembly language. The "PortASM" tool from MicroAPL converts 680x0 assembly language into PowerPC assembler.

Remember: we feel that the best solution is to re-write your assembly code as portable C.

Other Languages

Unless a third party steps in with a solution for a particular language, developers who use other than C, C++, Pascal, or Assembler have very few options. Your options are limited to re-coding in C/C++, using a binary translator, or running under emulation.

Using the RS/6000 development system

This lab will walk you through the process of compiling, linking, and running a PowerPC application using MPW's `remote` tool and an RS/6000.

Install MacTCP

If you don't have MacTCP installed, you'll need it for the `remote` tool. (We've already installed it for you.)

Mount the remote volume and set the current directory

Even though all of the compiling and liking will take place on an RS/6000, the MPW remote tool needs to "see" the target directory in order to select the appropriate directory on the UNIX machine. Therefore, you will need to mount the UNIX volume (if not already there) and set MPW's directory appropriately.

class note

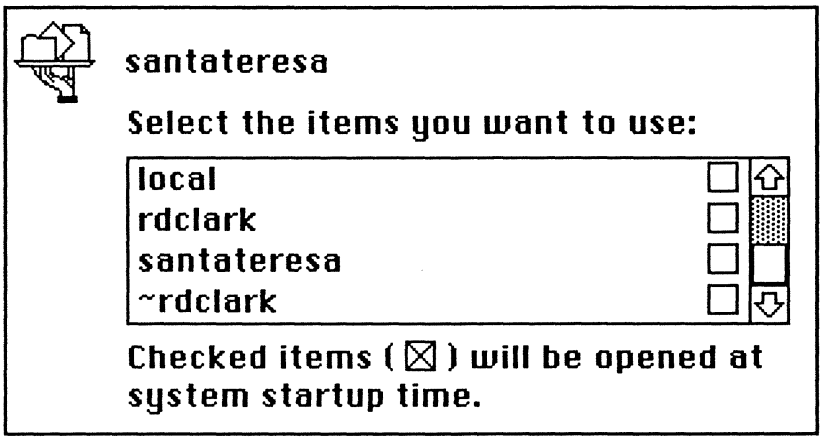
The class' server is located on "tomservo", which is located in the "Development Tools" zone on the Engineering net. You should log in as "student", with a password of "student". Mount the "student" volume, then set the directory to the folder indicated by your student number, e.g. "student:student1:". (Remember that the UNIX file server is case sensitive, so you have to supply the right case to MPW as well.)

class note

The UserStartup•MountMuslin file automatically mounts the RS/6000 and sets the current directory.

.afpvolumes and ~

By default, Helios (the AppleShare Filing Protocol server we're running on the RS/6000) exports your "home" UNIX directory as *~your-user-name* since ~ is the UNIX symbol for "home directory." Since MPW has problems with the ~ symbol in file and volume names, we've created an .afpvolumes file which re-exports your home directory as *your-user-name* without the ~. (See the following figure.) You can see this file by downloading it to your machine (from the server), removing the leading ".", and opening it in any text editor.



Set the remote machine, user, and port (in MPW)

Remote executes commands on the RS/6000 by sending TCP messages to a "Tool Dæmon" running on the remote machine. Remote uses two MPW shell variables to determine where the command should be sent, and one variable to control access to the remote system:

RemoteUser - Your UNIX user name
RemoteHost - The name of the remote machine
RemotePort - (optional) The port # on which ToolD is listening. This is only required if the port number is different than the default value of 5000. (We're using 5105)

(there is no "remote password" variable since having to mount the UNIX volume provides a reasonable degree of password protection.)

setting the variables

You can check to see if these variables are set by executing the MPW `set variable-name` command, which will echo the value of the corresponding variable.

```
set RemoteHost
set RemoteHost tomservo
```

If you need to set these variables, you can also use the `set` command, followed by an `export` command:

```
set RemoteHost tomservo ; export RemoteHost
```

determining the port number

If you are uncertain about the port number you need, contact your system administrator. You can also determine the port number by logging onto the UNIX system using Telnet, and executing the `ps -efa | grep ToolD` command (but make sure you get the case right!) The result should look like this:

```
$ ps -efa | grep ToolD
 rdclark 16045 16547   1 11:18:11  pts/3  0:00 grep ToolD
 rdclark 19556     1   0   Apr 29      -  0:00 /usr/bin/mac/ToolD 5104
 francis 19752     1   0   Apr 30      -  0:01 ToolD 6969
```

This shows that the Tool Dæmon for rdclark was started on port 5104, and the Tool Dæmon for francis was started on port 6969. If you see multiple copies of ToolD running on different ports, then you will need to pick your own port number > 5000. However, if you see a line that looks like:

```
root 19500     1   0   Apr 30      -  0:01 ToolD 5000
```

then there is probably a system-wide copy of ToolD which is shared by all users.

starting ToolD

If ToolD isn't running (as indicated by the ps command above or by an error when attempting to run remote), you will need to start it yourself or call your system administrator to start it.. You start ToolD yourself by logging in via Telnet, and executing the command:

```
ToolD port-number &
```

where port-number is a unique number you've selected. If you get an error stating that ToolD isn't found, you'll need to supply the full path:

```
/usr/bin/mac/ToolD port-number &
```

If this fails, call your system administrator for assistance.

class note

The variables should have the following values:

RemoteUser	student
RemoteHost	tomservo
RemotePort	*set if needed, i.e. the remote system doesn't have a single default port 5000. We're using 5105.*

These are already defined in "UserStartup•aStudentID" for you.

Try a remote command

Once you've set up the shell variables (and started ToolD, if necessary), you can try out a simple remote command. The Print Working Directory (pwd) command is as good as any, so try executing the following command in MPW:

```
remote pwd
```

If everything works, the beach ball will spin for a moment, and you should see something like:

```
/home/student
```

appear in the worksheet.

common error messages and their solutions

The most common errors come from setting the RemoteXXX variables incorrectly:

Cannot connect to port 5000... - You have either specified an incorrect port number, or ToolD is not running. Review the sections on "determining the port number" and "starting ToolD" above.

Cannot change directory to... - The most common cause of this is that you forgot to set MPW's default directory. Make sure that you have the volume which corresponds with your UNIX home directory mounted, and select that volume as MPW's default directory.

Build the application

Now comes the easy part. Since we've supplied a makefile, just select "Build..." from MPW's Build menu, and type in "Muslin". This will begin issuing commands to the RS/6000 via remote.

The commands you'll see executed include:

<u>UNIX commands</u>	
cmac	The C compiler, with "Macintosh compatibility" options set
ld	The linker
makepef	Converts XCOFF files to PEF files, and supplies names for some of the file-based libraries.
makesym	Create .SYM files from the XCOFF files
rm	Delete one or more intermediate files
strip	Remove the traceback tables (AIX debugging information) from XCOFF files. This is done to reduce the amount of disk storage needed, and in on way affects makesym.
<u>MPW commands</u>	
setfile	Set a file's creator, type, and finder flags
Rez	The resource compiler

ignore the warnings from ld

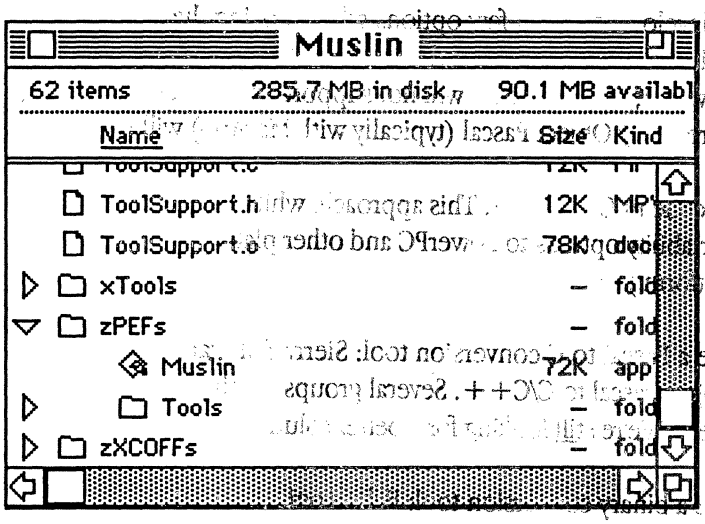
ld will issue several warnings about "import version of x replaced by import definition." You may ignore these, as they indicate the presence of some old (incomplete) glue code in the libraries being replaced by newer versions from a different library.

Transfer the application to the PowerPC Macintosh

Once you have compiled Muslin, you need to transfer it to your PowerPC machine. You can do this by mounting the UNIX file server from the PowerPC Macintosh and dragging the application across.

finding the application

Our makefile places the completed application in the zPEFs subfolder in the Muslin folder. (This folder should have a tools subfolder to hold the finished tools.) The zXCOFFs subfolder contains the intermediate XCOFF versions and the symbolic debugging files. The xTools subfolder contains the tool sources.



Run the application

Just go over to the PowerPC Macintosh and do it!

optional step

Build the tools & transfer them to the PowerPC Macintosh

The tool sources are located in the `xTools` subfolder, so you need to change the default directory before running remote. You can build all of the tools by executing the MPW shell script

```
AllTools.build
```

or you can build an individual tool using the command

```
AnyTool.build tool-name
```

as follows:

```
AnyTool.build Square    # Note: Do not type "SquareTool",
                          and look out for case
```

Most tools require the PolyUtils library, which can be built by running

```
PolyUtils.build
```

This is also created automatically by the `AllTools.build` script.

Ignore the warnings

The linker will complain while linking the tools since main doesn't represent the address of some code, but is a pointer to a data structure. This is a perfectly legal thing to do, as long as the application which loads the tools knows about it (as ours does.)

Converting THINK C code to PowerPC

This lab looks at the steps required to convert a simple (but old) THINK C application for use with the PowerPC compilers. The first part of the conversion is done in MPW, and the latter part is done on the RS/6000.

Create a .r file for use with Rez

If your THINK C projet uses a stand-alone resource file (as created by ResEdit or Resorcerer), you probably should convert this into a .r file. MPW's DeRez command can do this for you, using a command similar to the following:

```
DeRez myFile {Rincludes}Types.r > myFile.r
```

This command isn't perfect, as it won't handle the "new" System-7 dialog and window centering options properly, nor does it know about 'vers' resources. (Both of these resource types will appear as hexidecimal data in the output file.)

handling some special data types

You can tell DeRez to parse the new window and dialog positioning options by including `-d SystemSevenOrLater=1` on the command line, and include the definition for 'vers' resources by adding the file `{RIncludes}SysTypes.r` to the end of the file list (but before the ">" redirection character.)

Adding #include directives

The resource file that Derez generates isn't complete — it requires that you include "Types.r" (and possibly a definition for SystemSevenOrLater and SysTypes.r if you included those options on the command line.) Therefore, you should add the following lines to the start of the resource file:

```
#define SystemSevenOrLater 1    // only include this if
                                // you used the -d option
                                // mentioned above
#include "Types.r"
#include "SysTypes.r"           // only if you included
                                // "SysTypes.r" on the
                                // command line
```

Use the C compiler to locate basic portability problems

The code that we're supplying was written using an older version of THINK C. As such, it has one or more of the following problems which you will need to locate and fix. (Hint: Use the MPW C compiler with the -r option, and then fix the errors that arise. If you fix things in the order shown, life will be much simpler.)

1. Missing #include files
   Including <Types.h>, <QuickDraw.h>, and <Windows.h> is a good place to start
2. TRUE and FALSE should be in lower case (per MPW's Types.h)
3. Fix references to qd variables, for example: "thePort" becomes "qd.thePort", and "ltGray" becomes "qd.ltGray".
4. Add function prototypes for all functions, if missing

Build and test on the 68K machine

Select **Build:Create Build Commands...** in MPW to create a makefile for this application. Build and test the resulting application on the 68K machine. Once you are satisfied that the 68K version works, you can move over to PowerPC.

Transfer the code to the RS/6000

Once your 68K version works correctly, drag the entire folder over to your folder on the RS/6000, and then set the MPW directory to that folder.

Construct a makefile for the PowerPC version

To create a new makefile, duplicate the old one and rename it to "PowerPCApp.make", then change the target application's name to PowerPCApp within the makefile.

The build process is essentially the same (i.e. Rez, compile, link) except that the link step emits an XCOFF file and you need to include a call to makepef to convert the XCOFF file to a PEF file (which becomes the application once you add the usual window and menu resources and execute a SetFile to set the type and creator.)

Change the C and link commands

You'll need to modify the compile and link commands to use remote. Use your class notes or Muslin's makefile as a guide. The output of the link command should be PowerPCApp.xcoff.

Add makepef and makesym

While the PowerPC Macintosh can execute XCOFF files directly, you should convert XCOFF to PEF for speed reasons. You do this using the makepef tool on the RS/6000. Again, use Muslin's makefile as an example (and remove the reference to ToolSupport.xcoff.)

Update the .c file for PowerPC

If you try to build the application now, you'll get some compiler errors and linker errors. There are 2 small changes you'll need to make to the source code: Adding a declaration for the "qd" global variable, and casting all Pascal-style strings to type StringPtr.

The declaration for qd should be placed near the start of the file (but after the #includes) and look something like this:

```
#ifdef __powerc
      QDGlobals qd;
#endif
```

Look for callback pointers and convert for Mixed Mode

One of the other problems that you'll encounter is that the compiler complains about a call to TrackControl. (We're supplying a tracking callback for PageUp and PageDown.) The "cheap" way to fix this is to comment out place we're passing the ProcPtr and substitute NULL. The proper way to fix this is to supply a UniversalProcPtr for Mixed Mode, which you will do in a later lab. For now, just pass NULL as the last parameter to TrackControl.

Update the .r file for the PowerPC version

At this point, you should have an application with PowerPC code in the data fork, and a .r file listing the standard resources. However, the Process Manager doesn't know you've

built a PowerPC application, and needs to be informed of that fact. To do this, you have to add a special resource of type 'cfrg', ID 0.

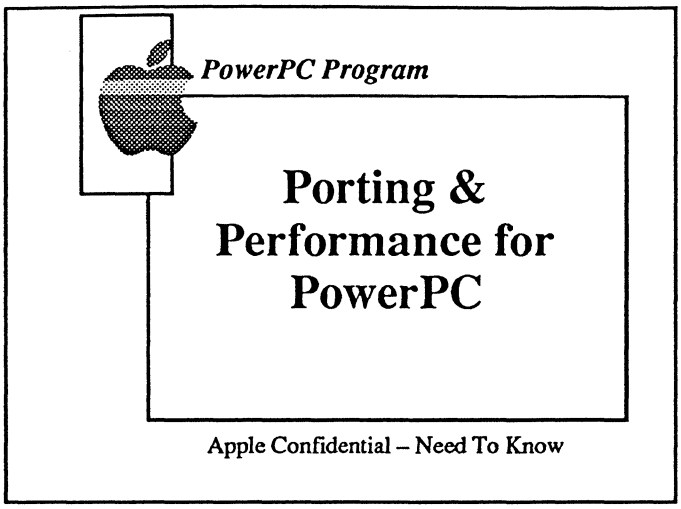
add a 'cfrg' resource

This resource's format is defined in CodeFragmentTypes.r (in the RIncludes folder), and you should enter the following definition into your resource file:

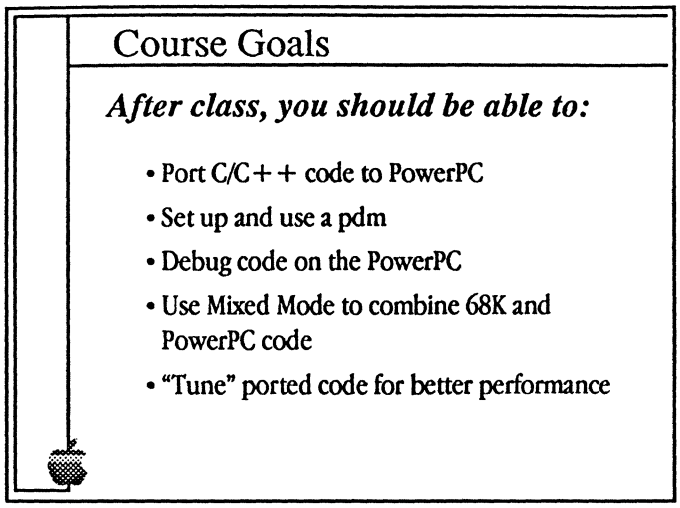
```
#include "CodeFragmentTypes.r"


resource 'cfrg' (0) {
   {
      kPowerPC,
      kFullLib,
        kNoVersionNum,kNoVersionNum,
        kIsApp,kOnDiskFlat,kZeroOffset,kWholeFork,
        "PowerPCApp"
   }
};
```

Build and test the PowerPC version

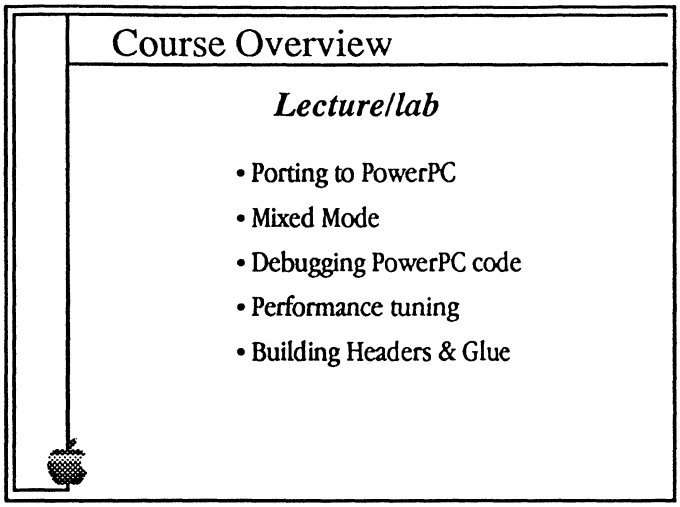
Fix any problems you encounter.



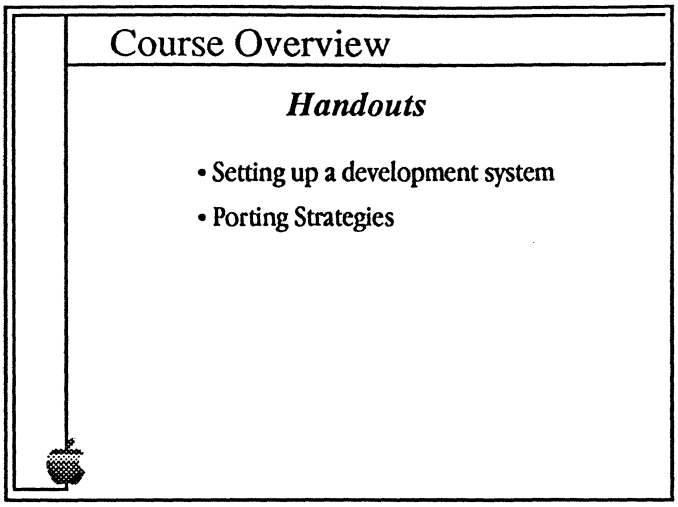
Porting & Performance for PowerPC



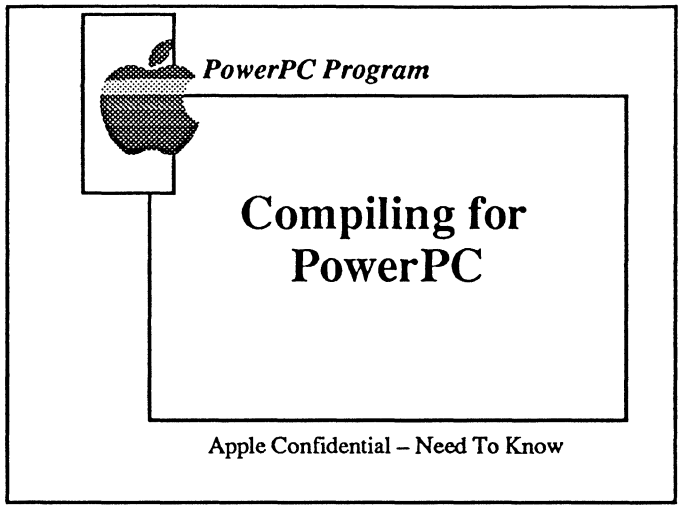
Course Goals



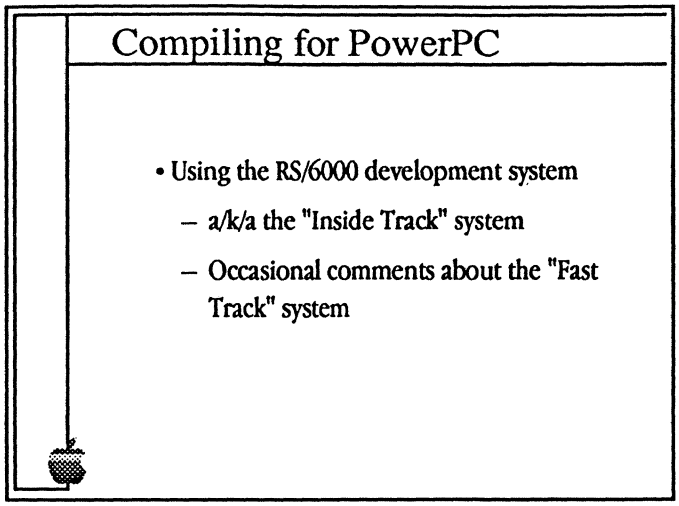
Course Overview



Course Overview

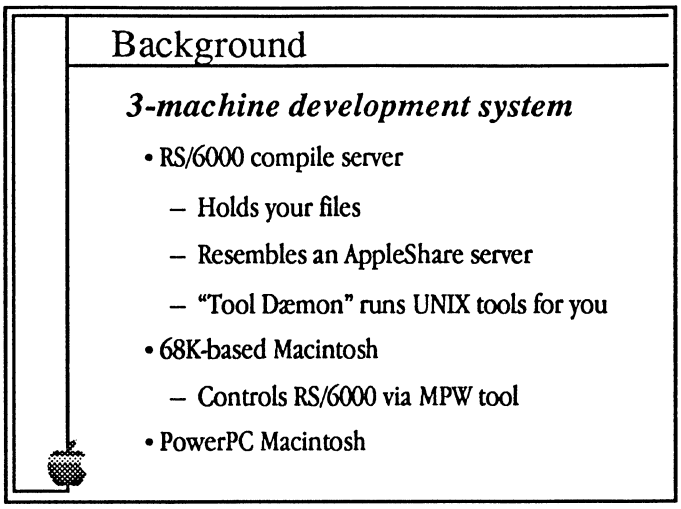


Compiling for PowerPC

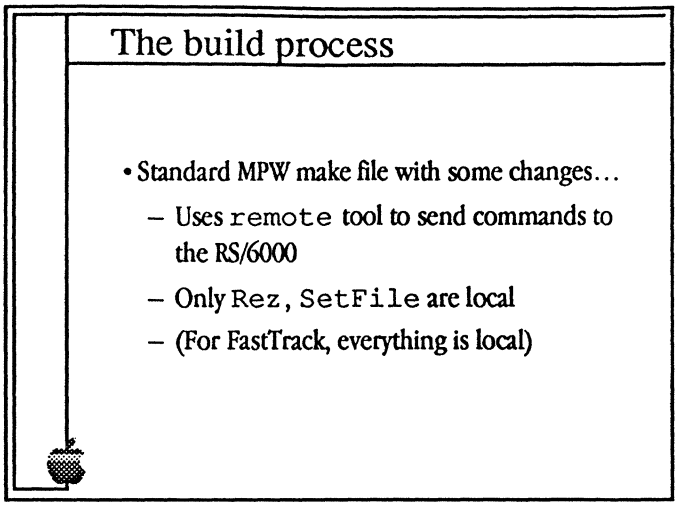


Compiling for PowerPC

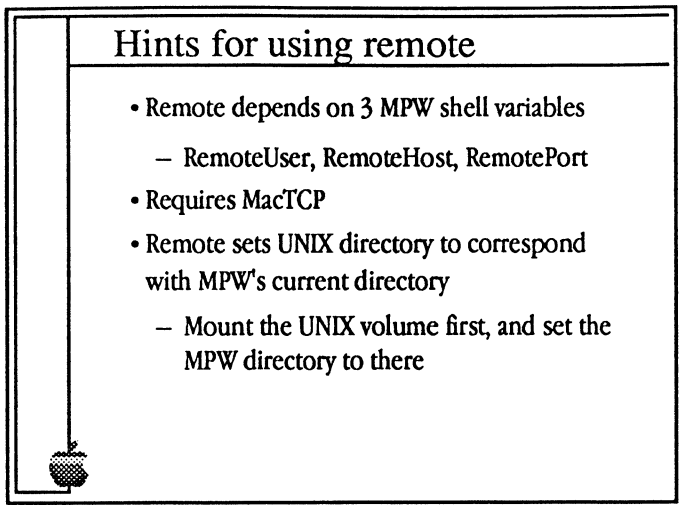
No A/UX system. Fast Track is for MPW on Mac OS.
It is not very stable now.


Background
3-machine development system
• RS/6000 compile server
– Holds your files
– Resembles an AppleShare server
– "Tool Dæmon" runs UNIX tools for you
• 68K-based Macintosh
– Controls RS/6000 via MPW tool
• PowerPC Macintosh

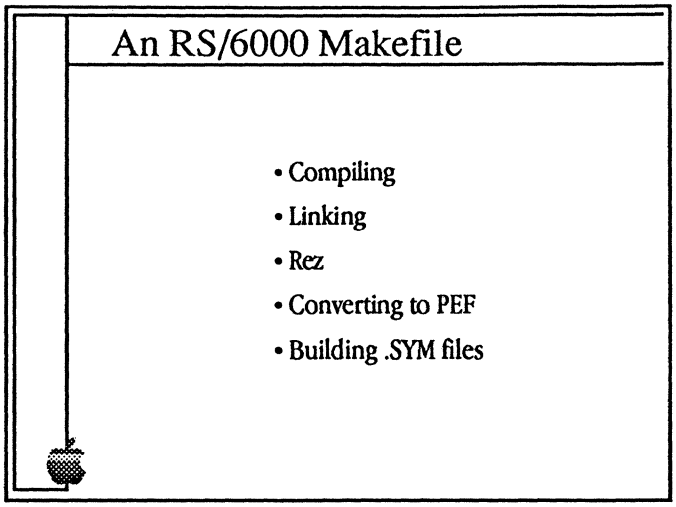
Background



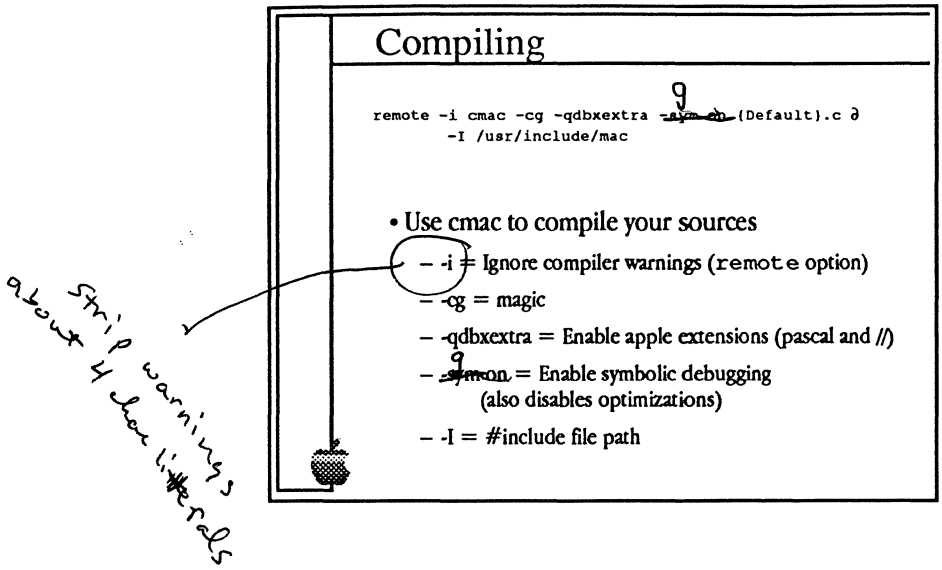
The build process



Hints for using remote

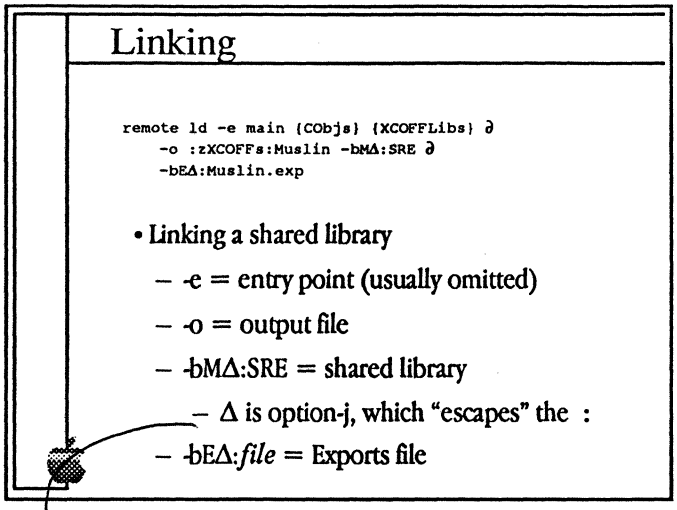


An RS/6000 Makefile



Compiling

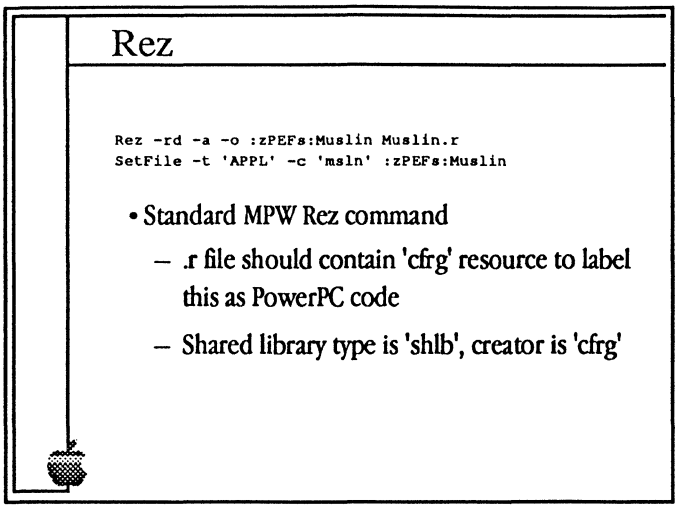
- Fast Track:
  PowerPCC -appleext on -sym on {default}.c -o {default}.c.o



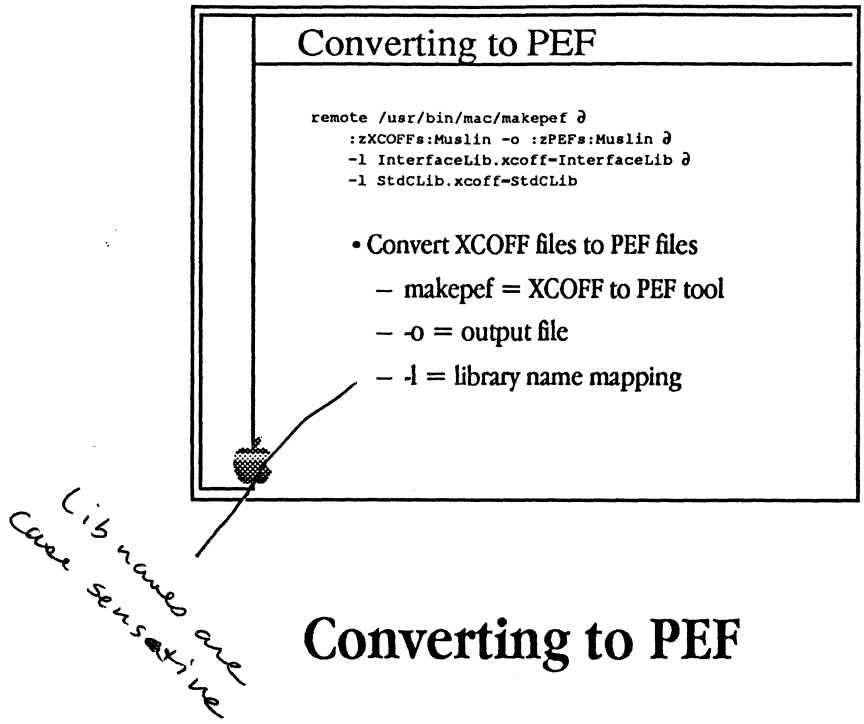
Linking

remote maps ":" into "Δ" unless escaped with Δ

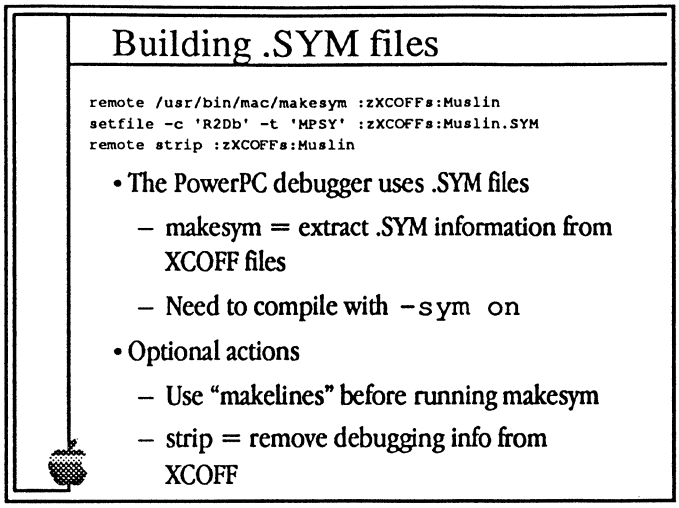
- Fast Track uses PowerPCLink
  - See release notes for current options


Rez
Rez -rd -a -o :zPEFs:Muslin Muslin.r
SetFile -t 'APPL' -c 'msln' :zPEFs:Muslin
• Standard MPW Rez command
– .r file should contain 'cfrg' resource to label this as PowerPC code
– Shared library type is 'shlb', creator is 'cfrg'

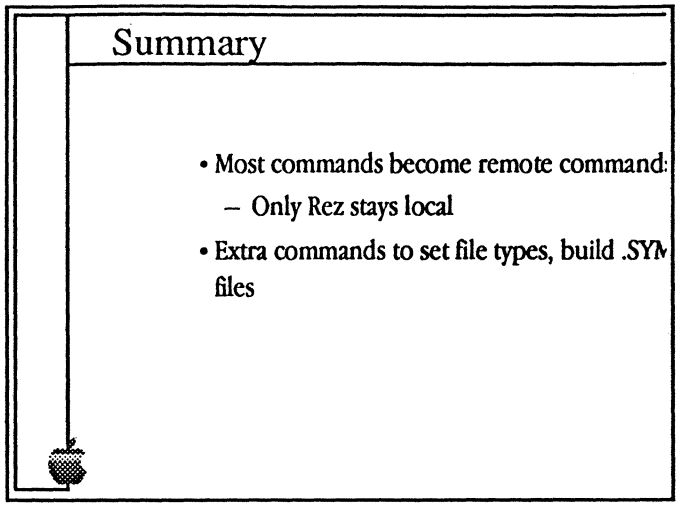
Rez



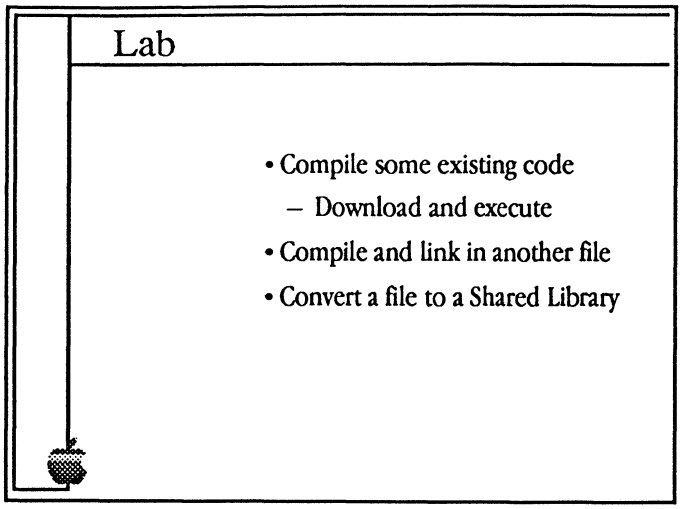
- Makepef also exists as a FastTrack tool on the 68K



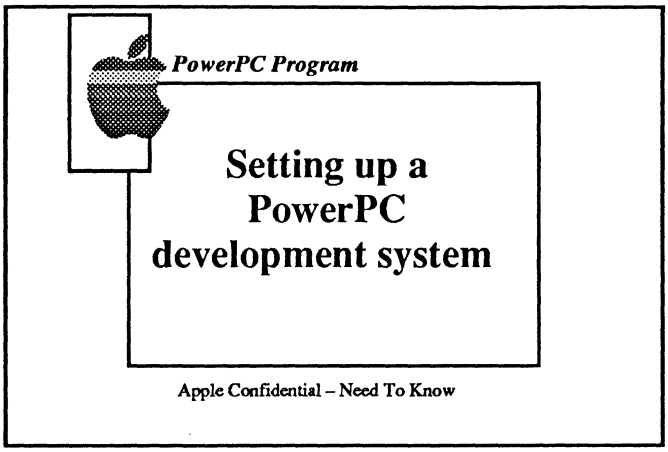
Building .SYM files


Summary
• Most commands become remote command
– Only Rez stays local
• Extra commands to set file types, build .SYM
files

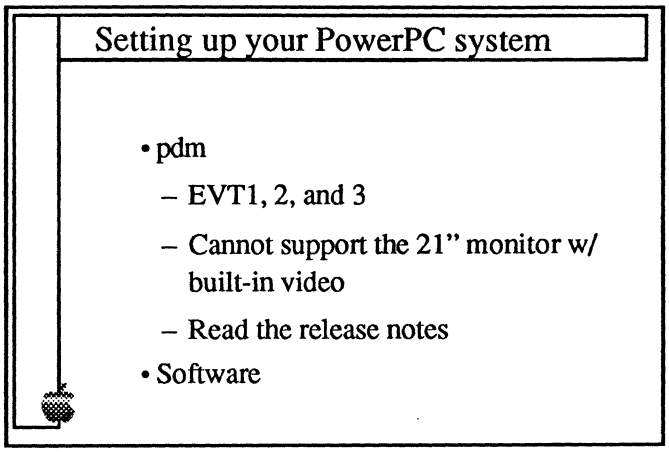
Summary


Lab
• Compile some existing code
– Download and execute
• Compile and link in another file
• Convert a file to a Shared Library

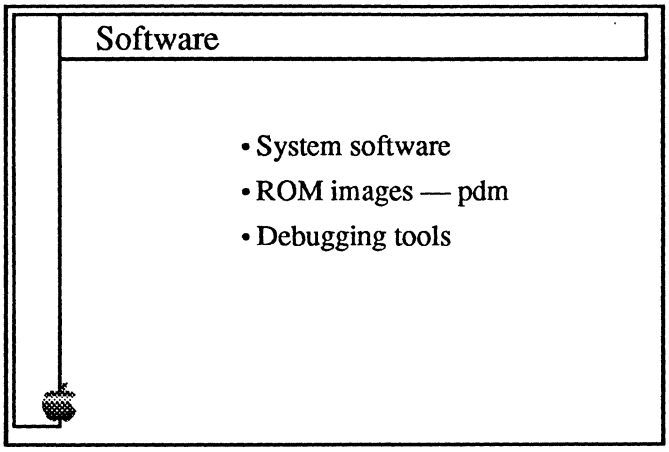
Lab



Setting up a PowerPC development system

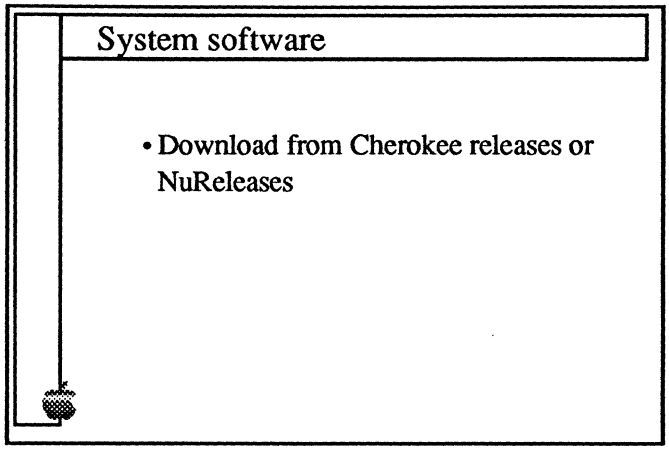


Setting up your PowerPC system

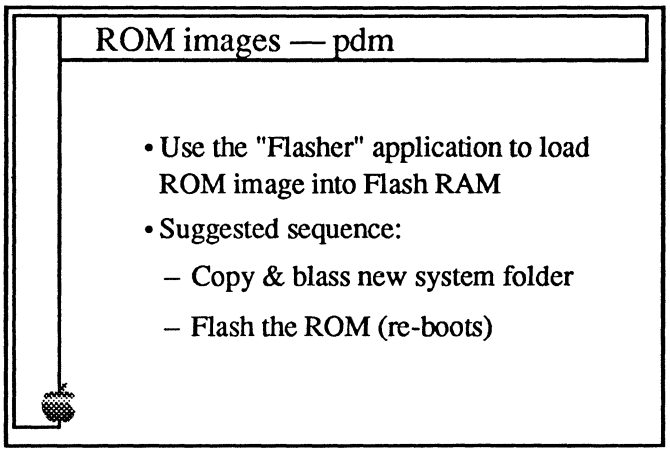


Software

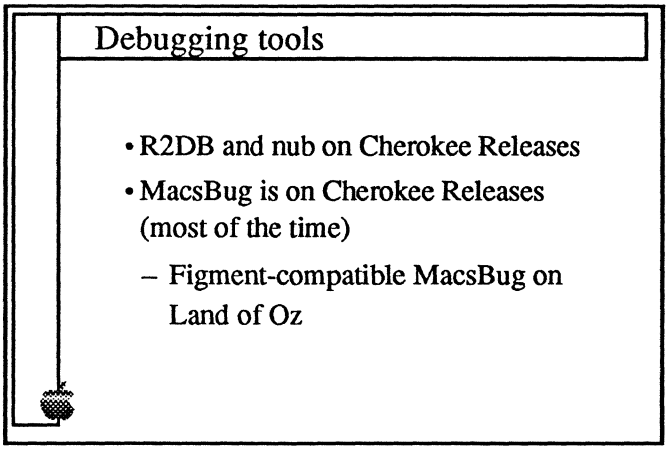
- System software available from Grand Cherokee



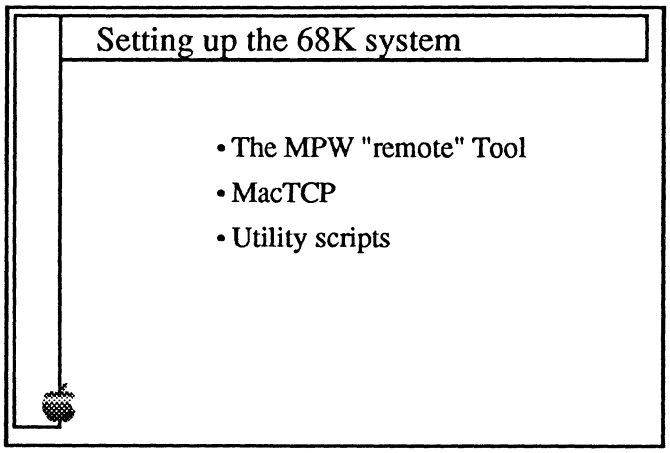
System software



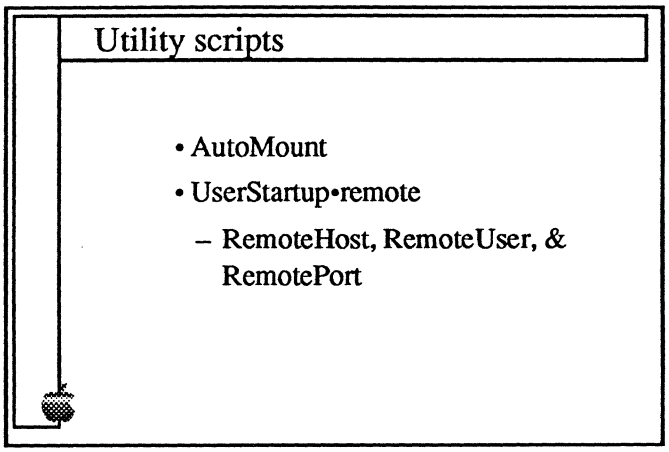
ROM images — pdm



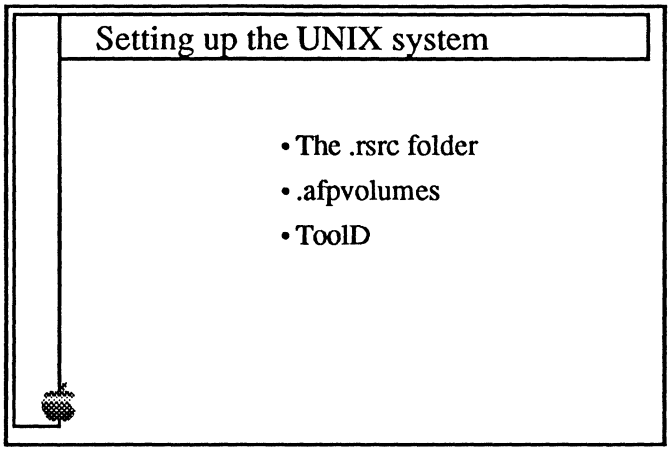
Debugging tools



Setting up the 68K system



Utility scripts



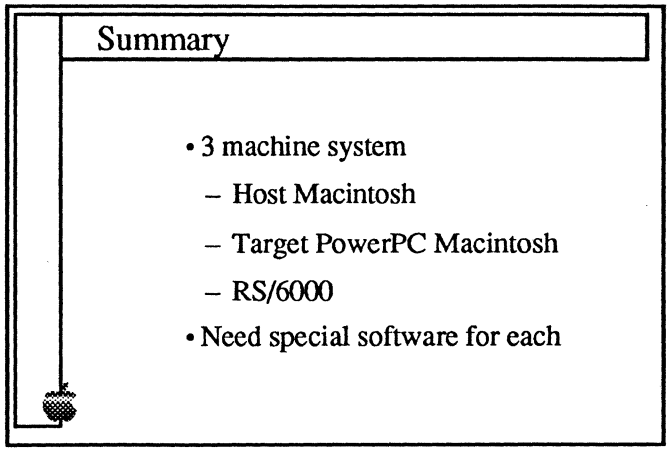
Setting up the UNIX system

Sample .afpvolumes line:

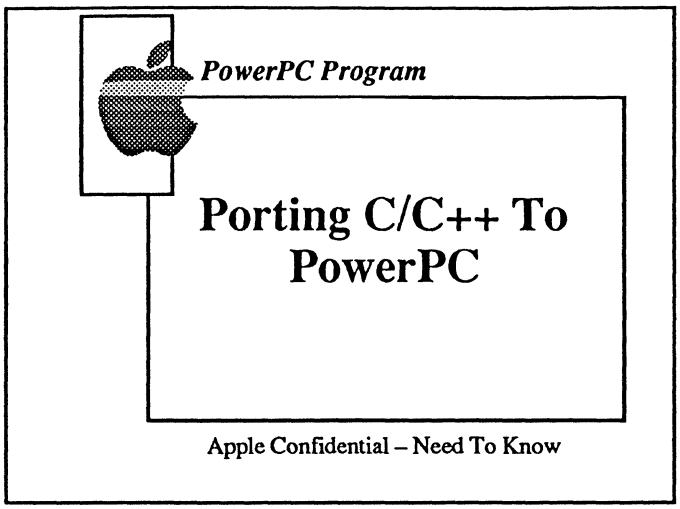
```
/home/santateresa/rdclark:rdclark::fixed:readwrite
```

Sample ToolD command

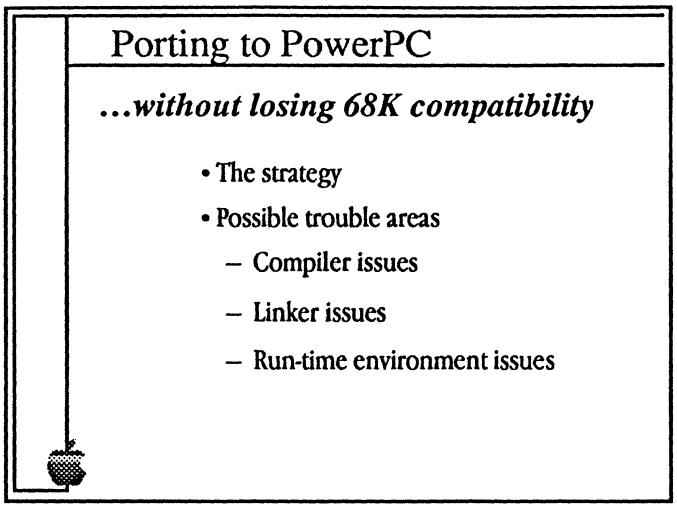
```
/usr/bin/mac/ToolD 5101 &
```


Summary
• 3 machine system
– Host Macintosh
– Target PowerPC Macintosh
– RS/6000
• Need special software for each

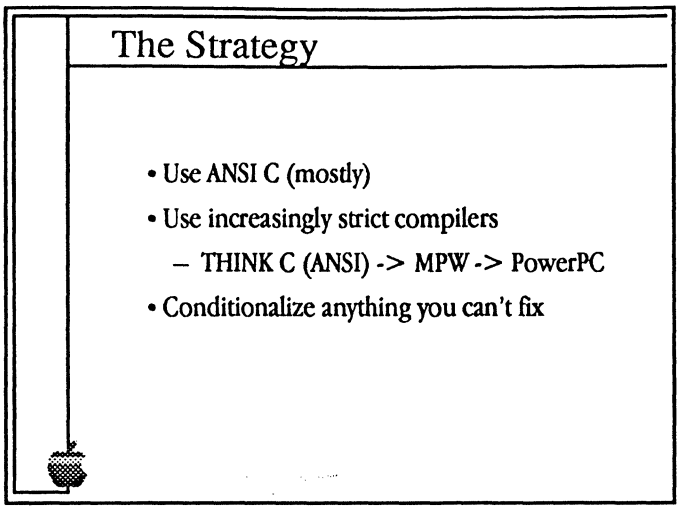
Summary



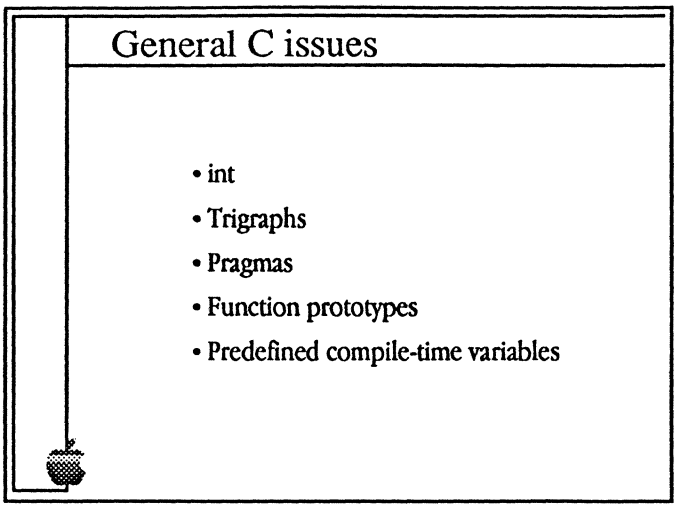
Porting C/C++ To PowerPC



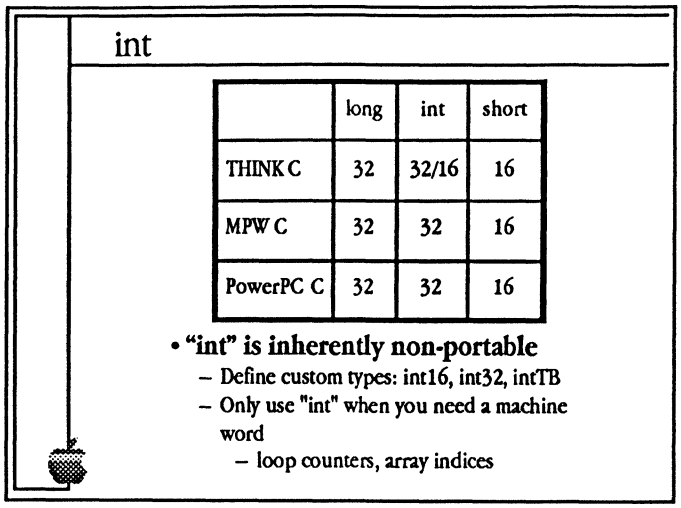
Porting to PowerPC



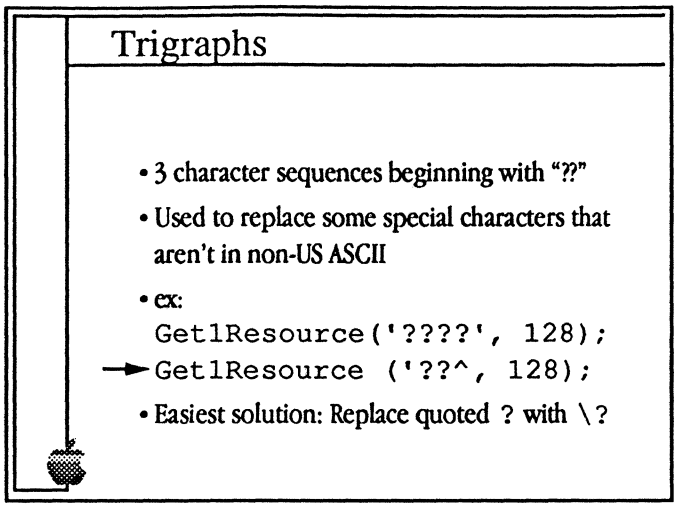
The Strategy



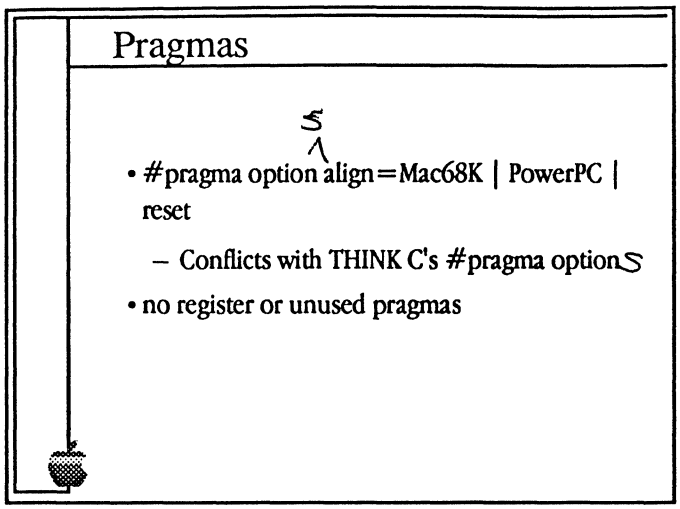
General C issues


int
long
int
short
THINK C
32
32/16
16
MPW C
32
32
16
PowerPC C
32
32
16
• "int" is inherently non-portable
– Define custom types: int16, int32, intTB
– Only use "int" when you need a machine word
– loop counters, array indices

int


Trigraphs
• 3 character sequences beginning with "??"
• Used to replace some special characters that aren't in non-US ASCII
• ex:
GetlResource('????', 128);
GetlResource ('??^, 128);
• Easiest solution: Replace quoted ? with \?

Trigraphs



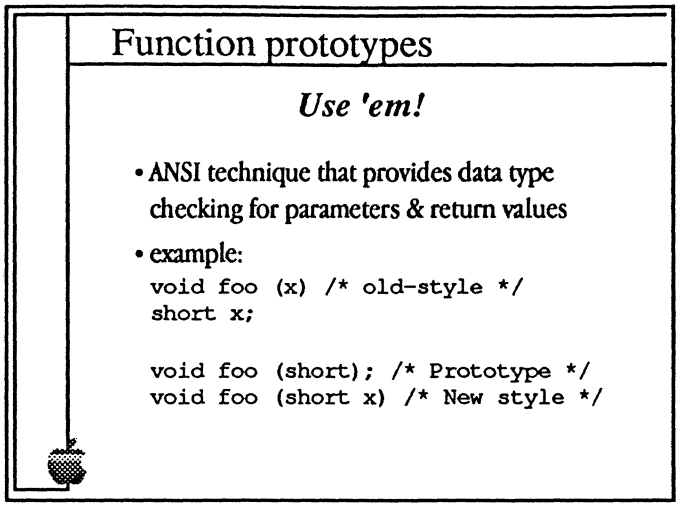
Pragmas

Struct alignment

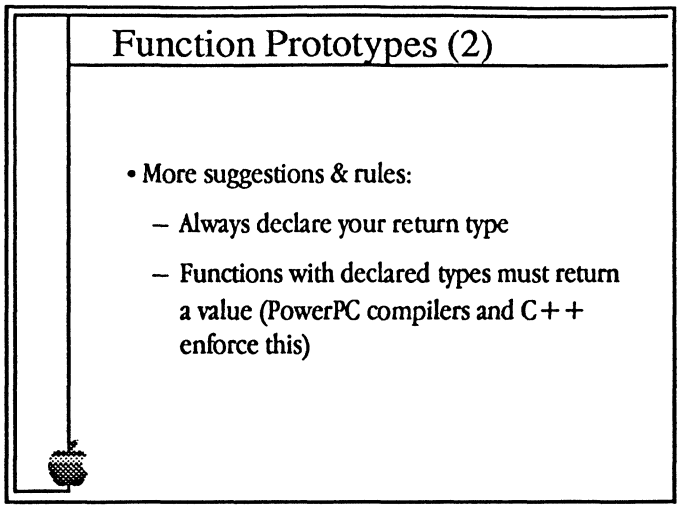
```
struct E
    short a
    long  b
};
```

68K = 6 bytes

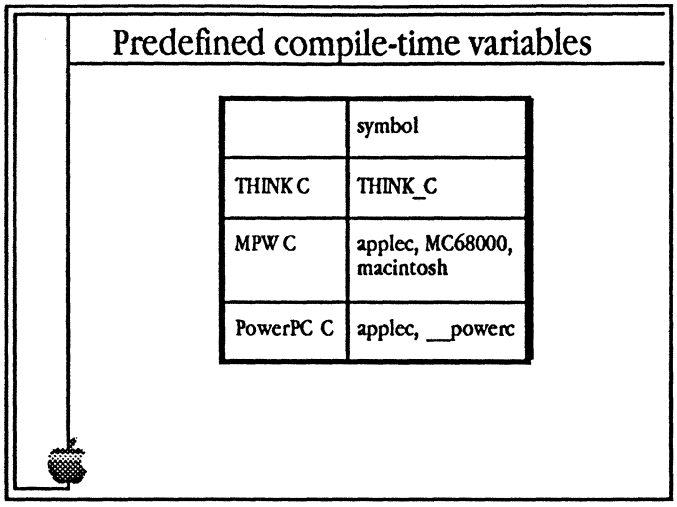
PowerPC = 8 bytes



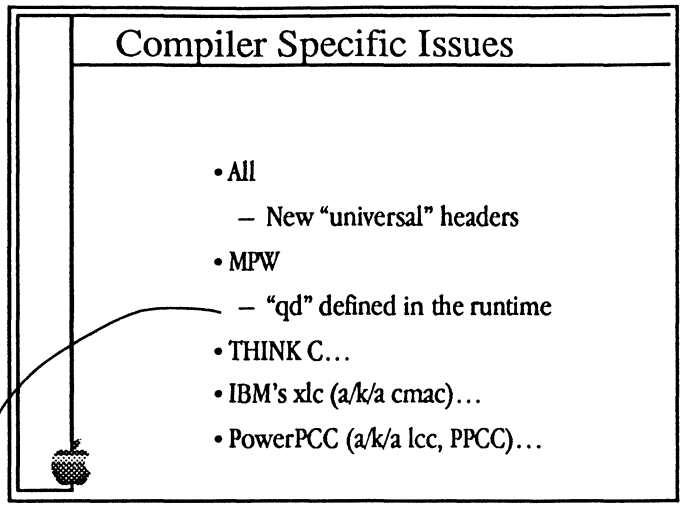
Function prototypes



Function Prototypes (2)

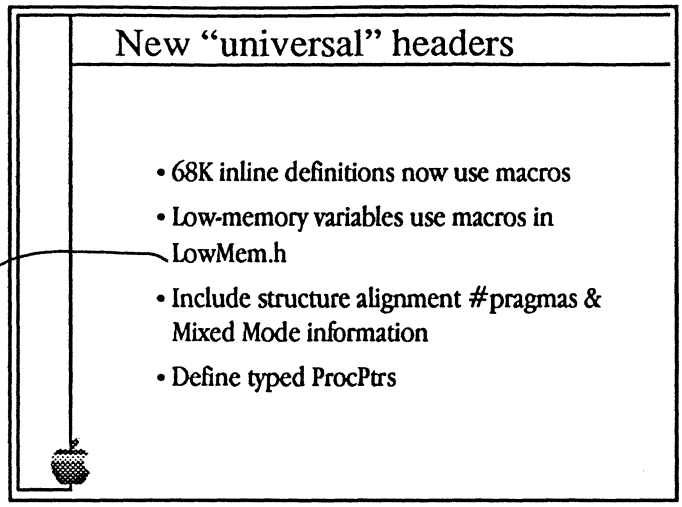


Predefined compile-time variables



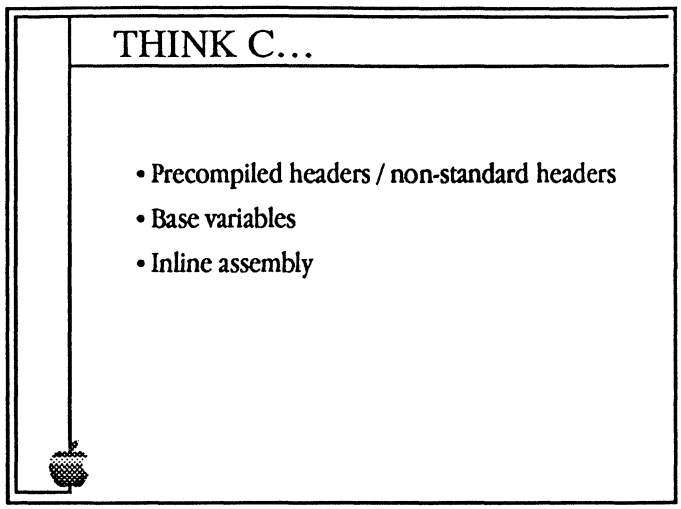
Compiler Specific Issues

```
#ifdef __powerc
    QDGlobals qd;
#endif
```



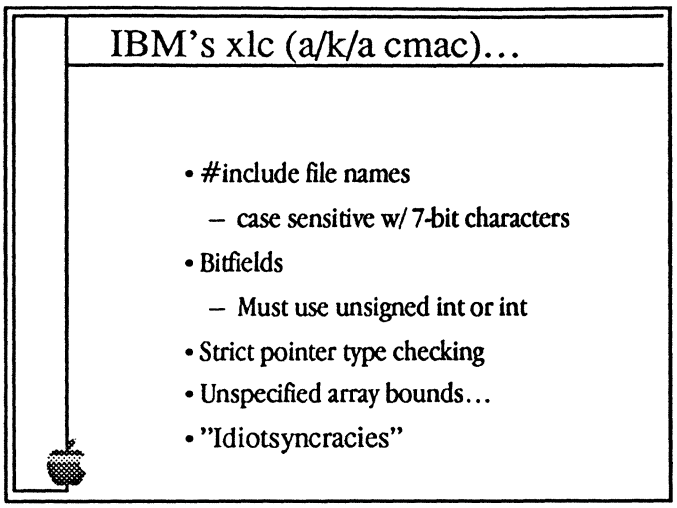
New "universal" headers

ResError()
MemError() > defined in LowMem.h

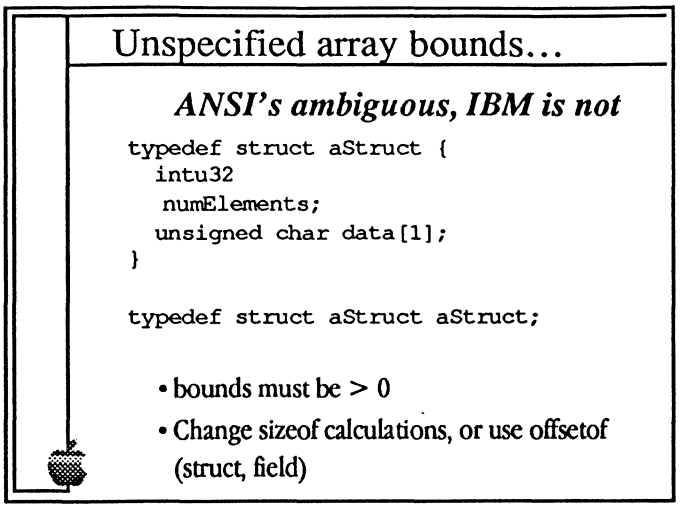


THINK C...

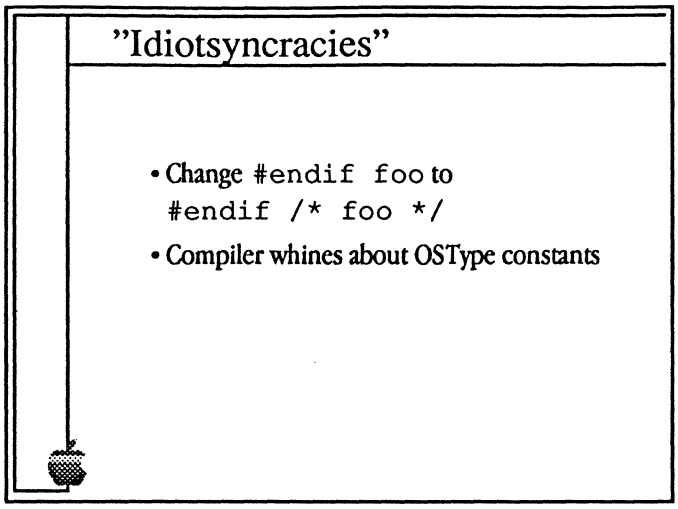
Look in New Devices.h and see funct prototype for ~~code~~ ioCompletion



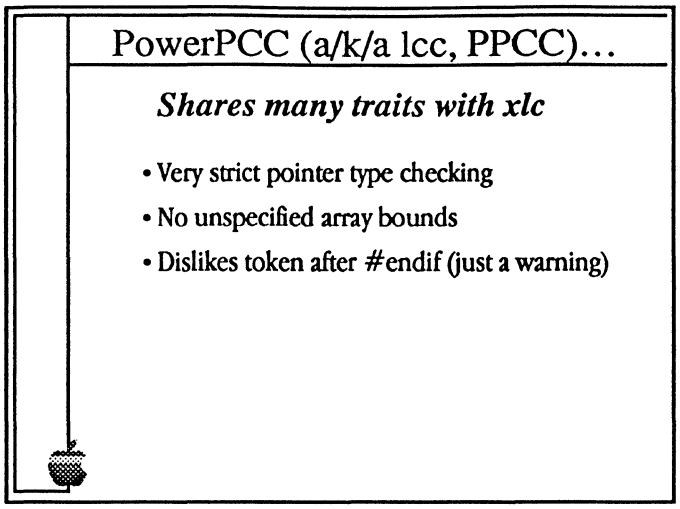
IBM's xlc (a/k/a cmac)...



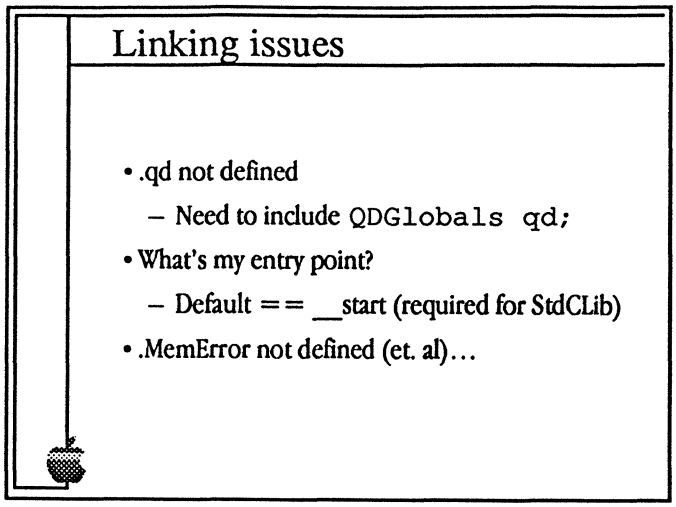
Unspecified array bounds...



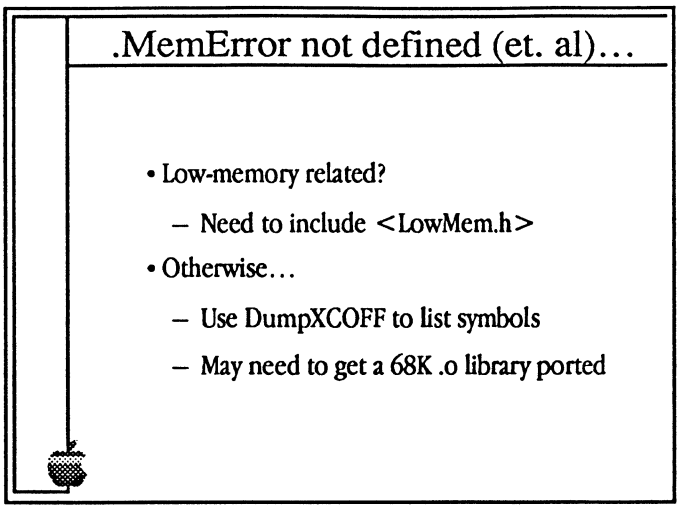
"Idiotsyncracies"



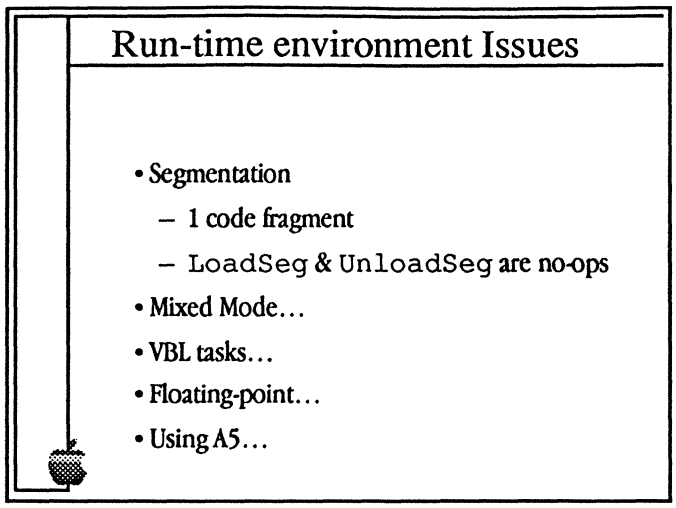
PowerPCC (a/k/a lcc, PPCC)...



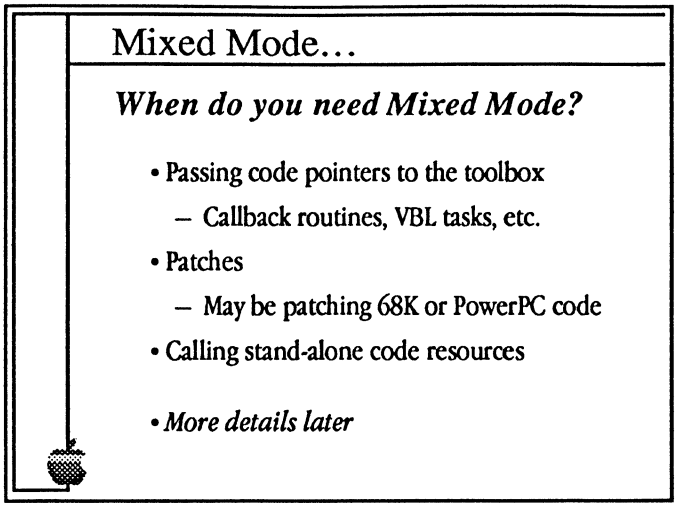
Linking issues



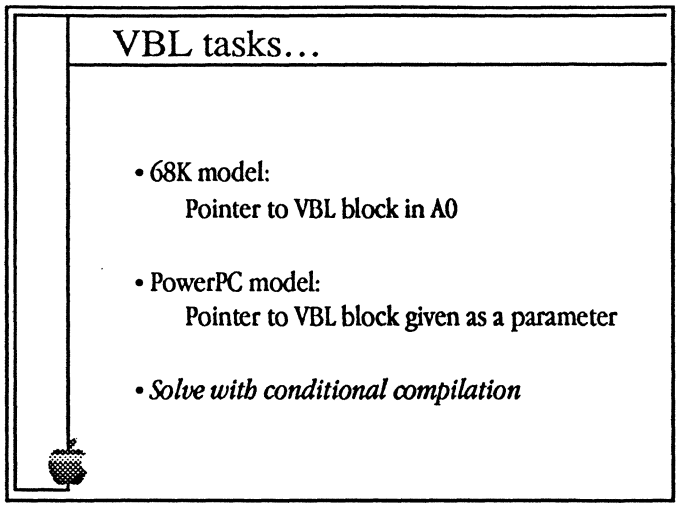
.MemError not defined (et. al)...



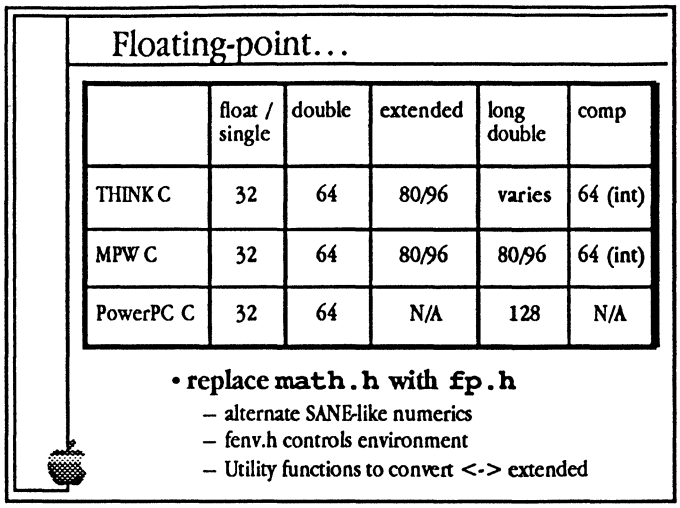
Run-time environment Issues


Mixed Mode...
When do you need Mixed Mode?
• Passing code pointers to the toolbox
– Callback routines, VBL tasks, etc.
• Patches
– May be patching 68K or PowerPC code
• Calling stand-alone code resources
• More details later

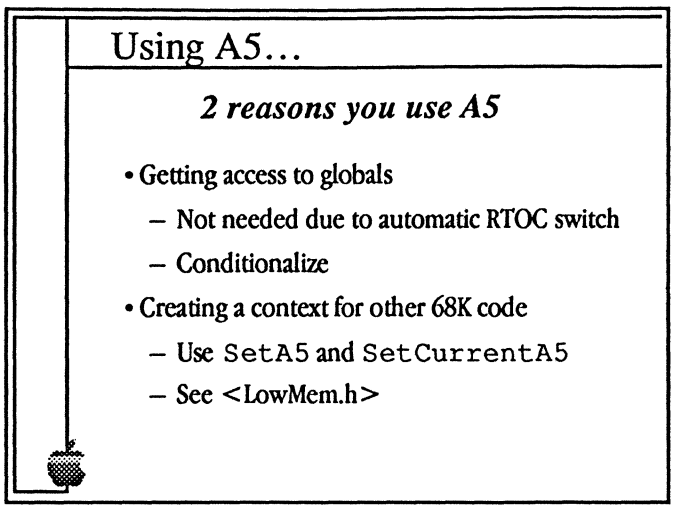
Mixed Mode...


VBL tasks...
• 68K model:
Pointer to VBL block in A0
• PowerPC model:
Pointer to VBL block given as a parameter
• Solve with conditional compilation

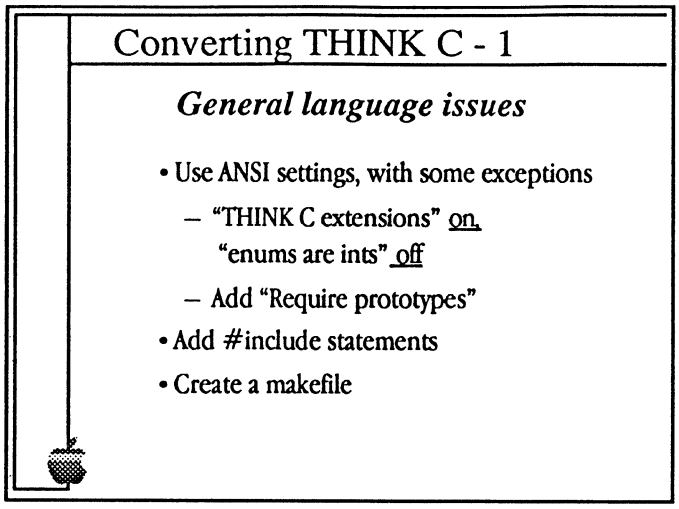
VBL tasks...



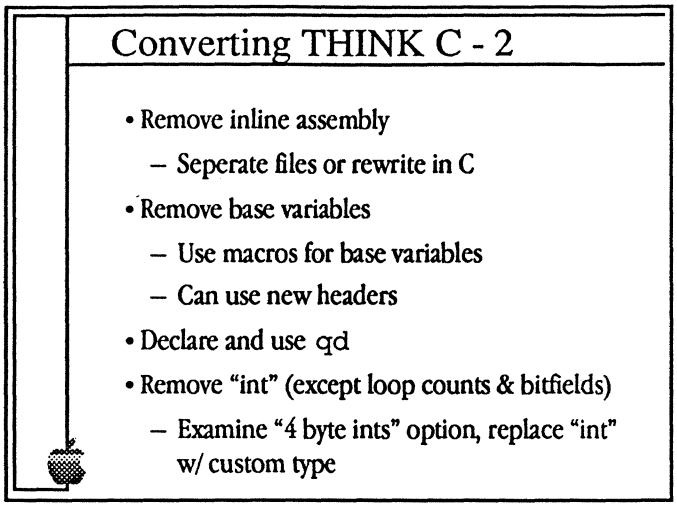
Floating-point...


Using A5...
2 reasons you use A5
• Getting access to globals
– Not needed due to automatic RTOC switch
– Conditionalize
• Creating a context for other 68K code
– Use SetA5 and SetCurrentA5
– See <LowMem.h>

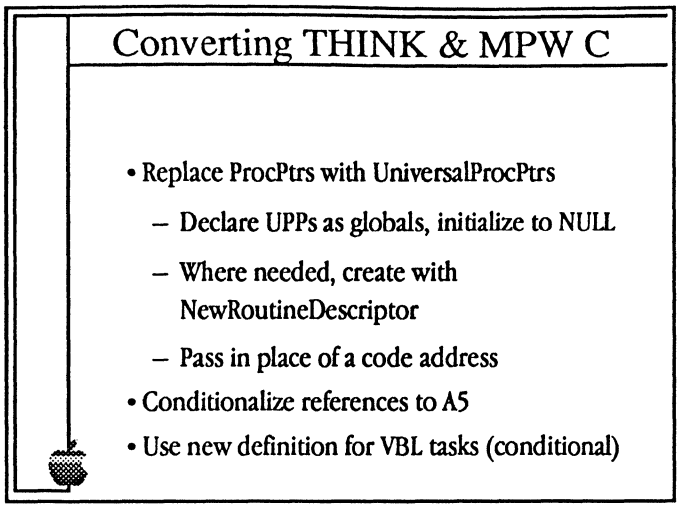
Using A5...



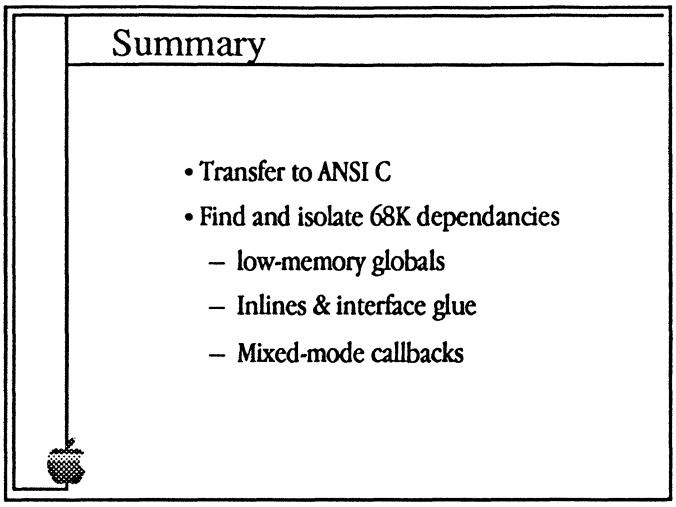
Converting THINK C - 1



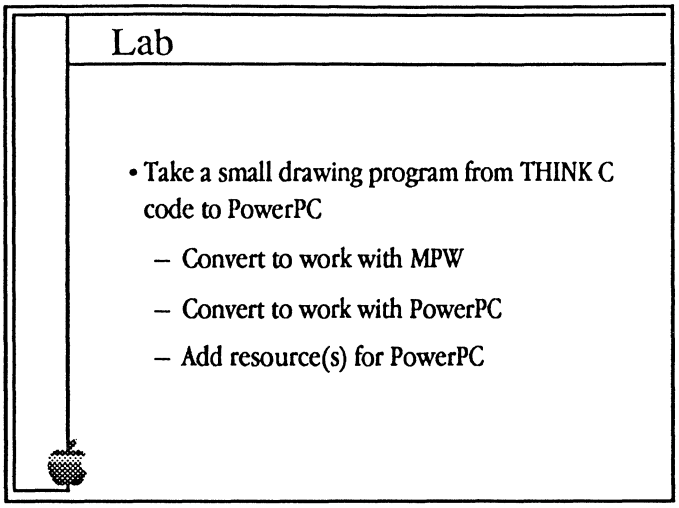
Converting THINK C - 2



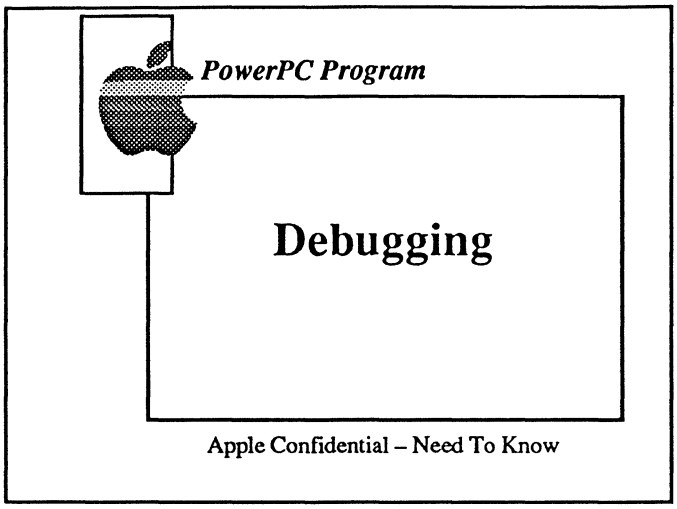
Converting THINK & MPW C


Summary
• Transfer to ANSI C
• Find and isolate 68K dependancies
– low-memory globals
– Inlines & interface glue
– Mixed-mode callbacks

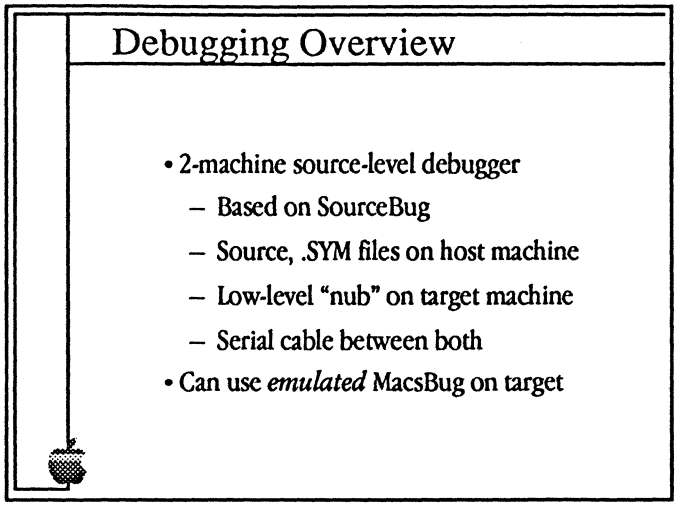
Summary


Lab
• Take a small drawing program from THINK C code to PowerPC
– Convert to work with MPW
– Convert to work with PowerPC
– Add resource(s) for PowerPC

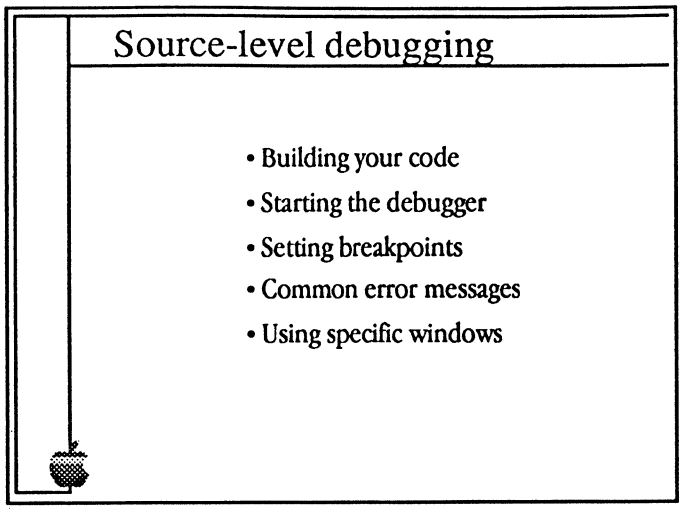
Lab


PowerPC Program
Debugging
Apple Confidential – Need To Know

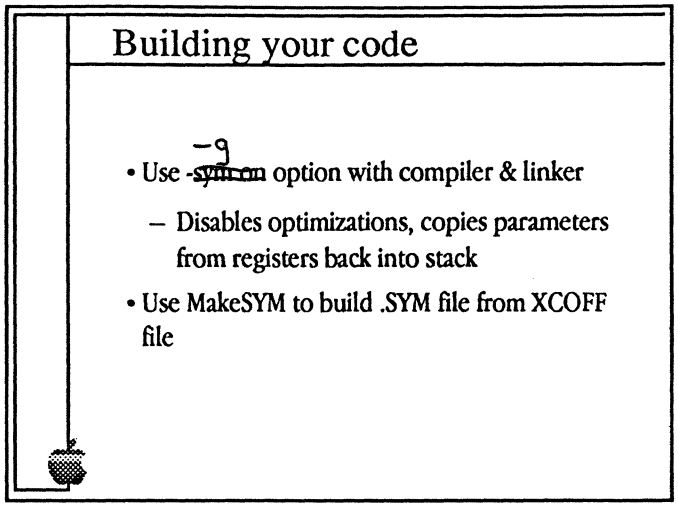
Debugging



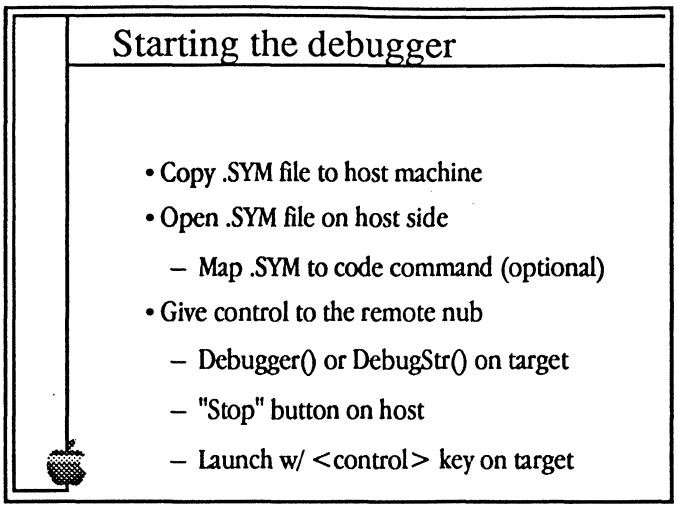
Debugging Overview



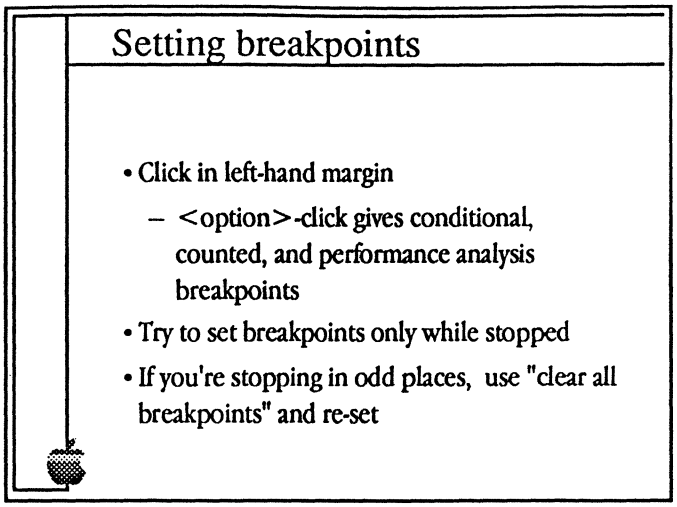
Source-level debugging



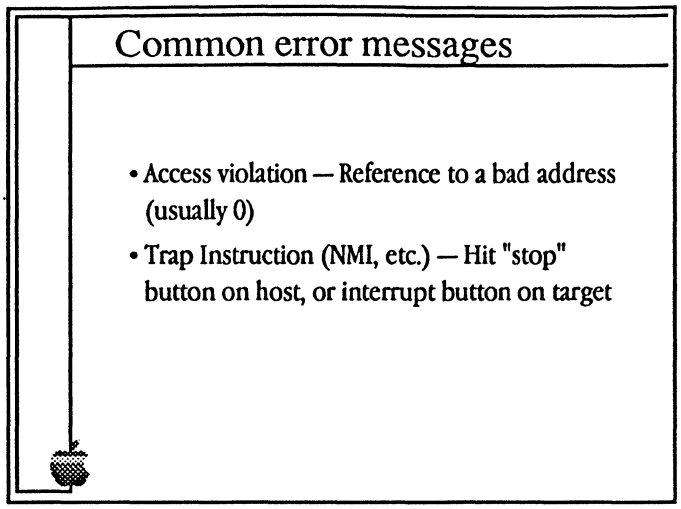
Building your code



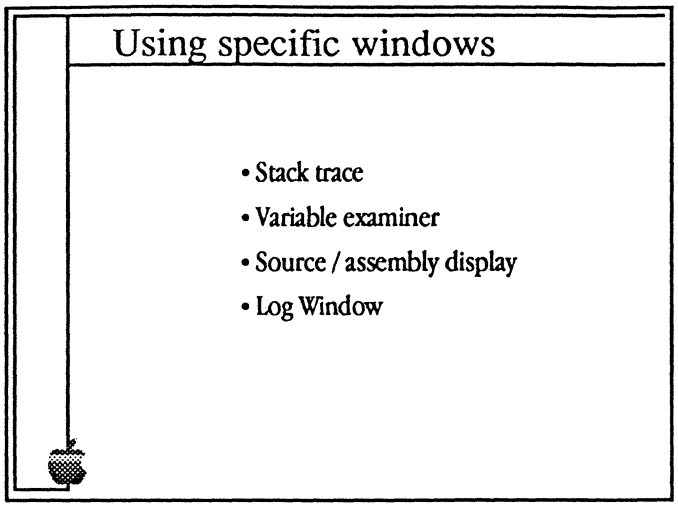
Starting the debugger



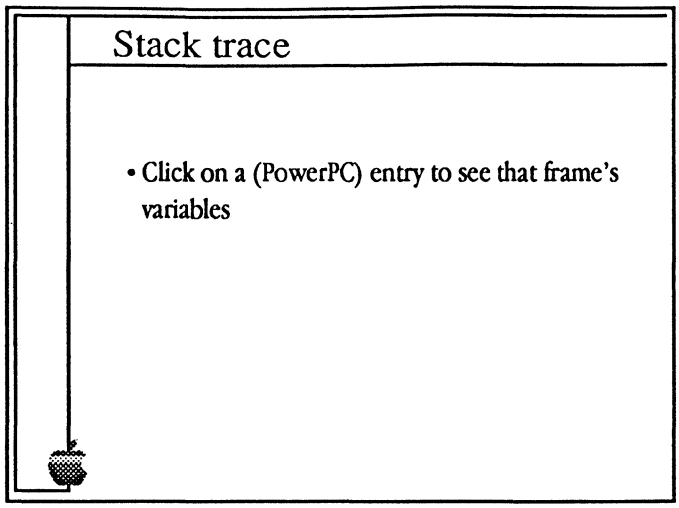
Setting breakpoints



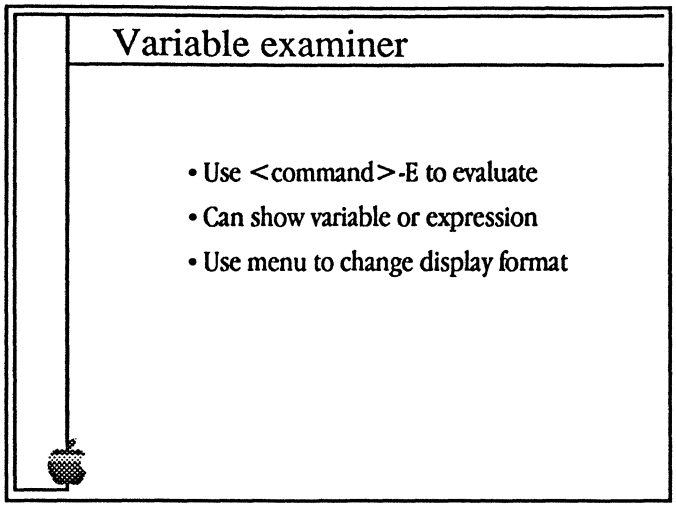
Common error messages



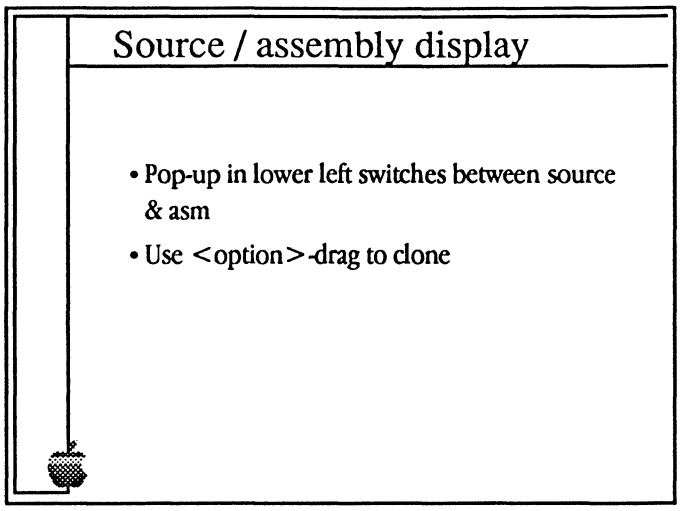
Using specific windows


Stack trace
• Click on a (PowerPC) entry to see that frame's variables

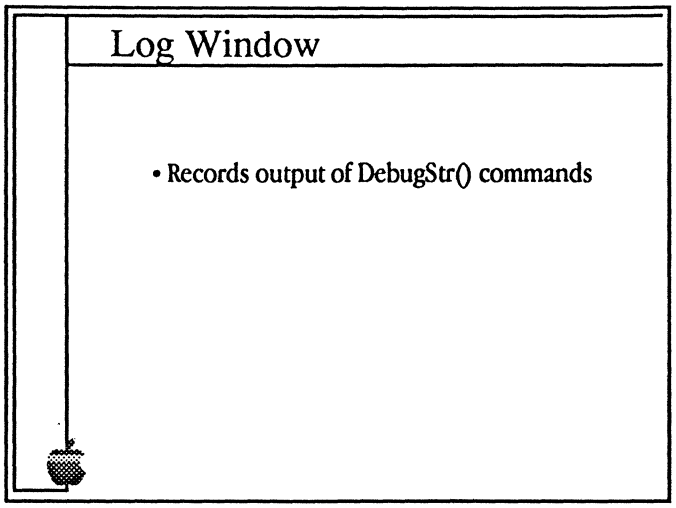
Stack trace



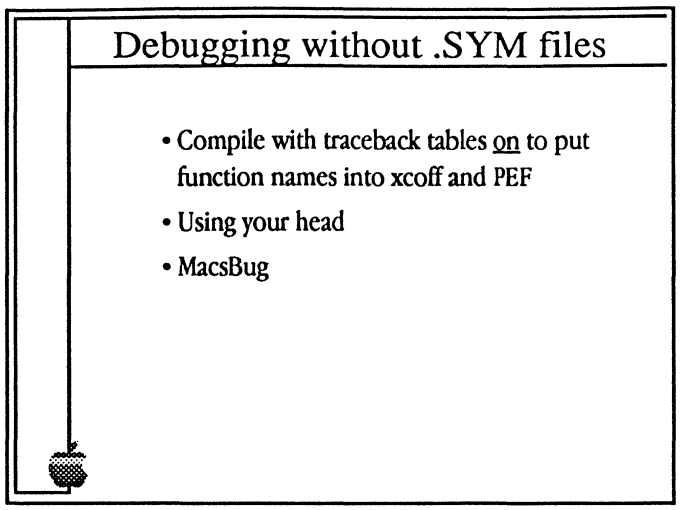
Variable examiner



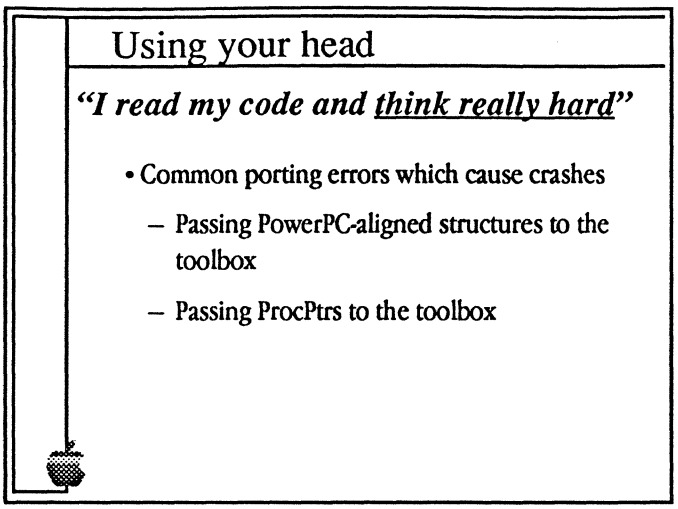
Source / assembly display



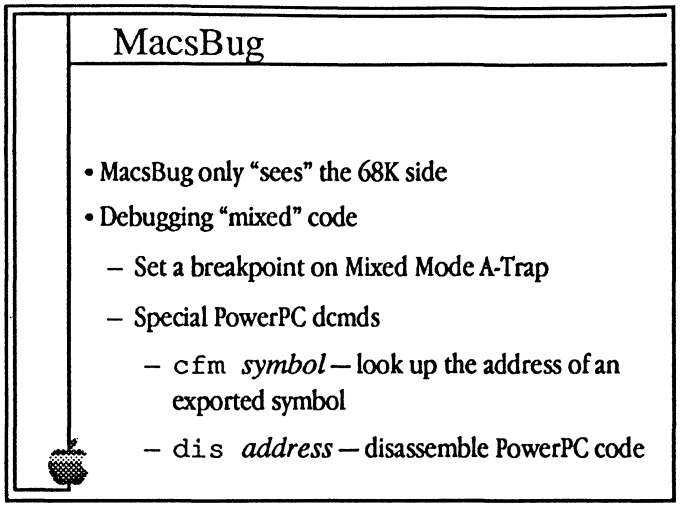
Log Window



Debugging without .SYM files

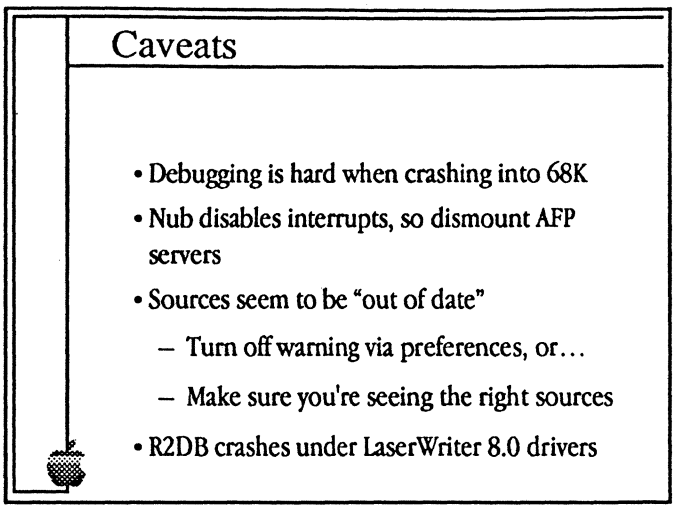


Using your head

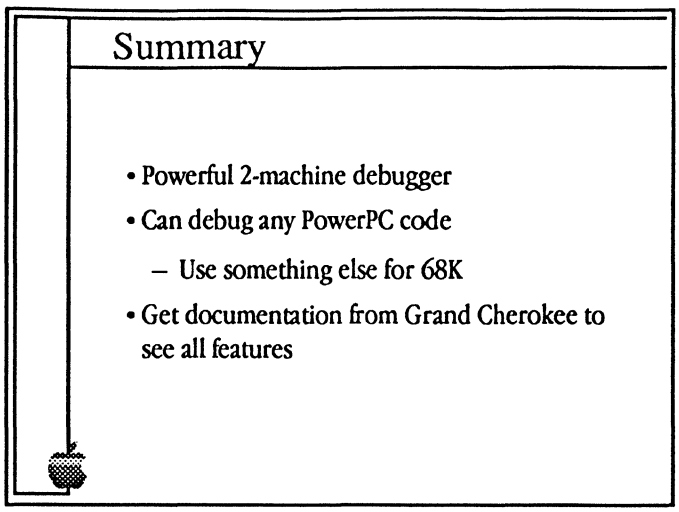


MacsBug

Scp - PowerPC stack frame dump

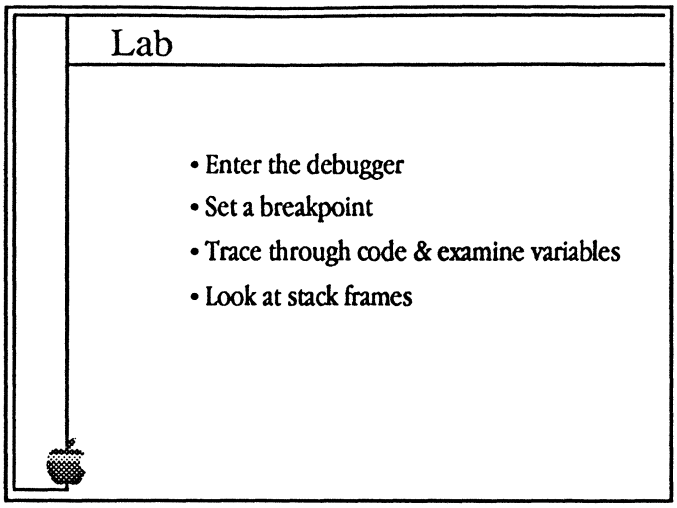

Caveats
• Debugging is hard when crashing into 68K
• Nub disables interrupts, so dismount AFP servers
• Sources seem to be "out of date"
– Turn off warning via preferences, or...
– Make sure you're seeing the right sources
• R2DB crashes under LaserWriter 8.0 drivers

Caveats



Summary

Nancy Lamn for access to doc for RocknRoll


Lab
• Enter the debugger
• Set a breakpoint
• Trace through code & examine variables
• Look at stack frames

Lab

Debugging with R2DB

This lab walks you through the process of debugging an application using the PowerPC 2-machine debugger R2DB.

Prerequisites

You should have built your copy of the application once already. If you haven't, get a copy of the lab application and .SYM files from the instructor.

Connect the host and target machines

The debugger communicates between the 2 machines through a serial cable. Since the Printer port is normally used for a LocalTalk connection, we'll connect the 2 machines through the Modem ports.

class note

The machines are already set up, but you should note the position of the cable on the pdm. Since the pdm machines are packaged in spare Centris 610 cases, the ports on the motherboard don't match up with the labels on the case. So, take a look and remember where the cable goes — this will save you some grief later.

Configure the nub on the target machine

The debugger nub on the pdm can use either the Printer or Modem ports for communicating with the 68K Macintosh. Use the "Debugger Nub Controls" control panel to connect the nub to the Serial port, and then re-boot. (Note: you only have to do this once, as the setting gets recorded in a special preferences file.)

Build and download the .SYM files

The debugger uses MPW-style .SYM files which are derived from the XCOFF files created by the compiler via the makesym UNIX tool.

class note

The supplied makefile and tool building scripts already do this for you.

Since the .SYM files are fairly large, you should download them to your host 68K Macintosh before using them with the debugger. (Note: You only have to download the .SYM files you actually need. If you're only debugging Muslin, just download Muslin.SYM.)

Keep those source files handy!

The debugger will need to read the source files in order to provide source-level debugging, so you should either download the sources to the 68K Macintosh, or leave the server volume mounted there. (Either way works. The advantage of leaving the server mounted is that you always have the latest version. The disadvantage is, obviously, speed.)

Launch the debugger (on the 68K Macintosh)

The easiest way to launch the debugger is by dragging a .SYM file on to it, or by double-clicking the .SYM file (assuming that you set the type and creator in the previous step.) Otherwise, you can launch the debugger application directly and use the **File:Open...** menu item to select a .SYM file.

Selecting the debugging port

The debugger will ask you which port has the debugger cable. You can set this value and eliminate the dialog by selecting **Edit:Preferences...** and setting the connection port to be the Modem Port.

Launch the target application & enter the debugger (on the PowerPC Macintosh)

Now, launch the Muslin application on the PowerPC machine. (It doesn't actually matter if you start the debugger before running the target application.) But, before you do anything else, read the next paragraph!

Dismount all servers!

Before doing anything on the pdm that could cause a break into R2DB, make sure you've dismounted all AFP server volumes! The nub is a very low-level debugger that disables interrupts on the target machine (just as MacsBug does.) This will cause the AppleShare code to miss one or more "tickle packets", and your machine could hang for a very long time waiting for the connections to time out. So, make sure you dismount those servers before executing the next step!

Enter the debugger

Your code can enter the debugger in several ways, either deliberately (via a call to Debugger() or DebugStr()), or inadvertently through an illegal instruction, bad memory reference, or a host of other causes. We'll take the deliberate route, and make a call to DebugStr().

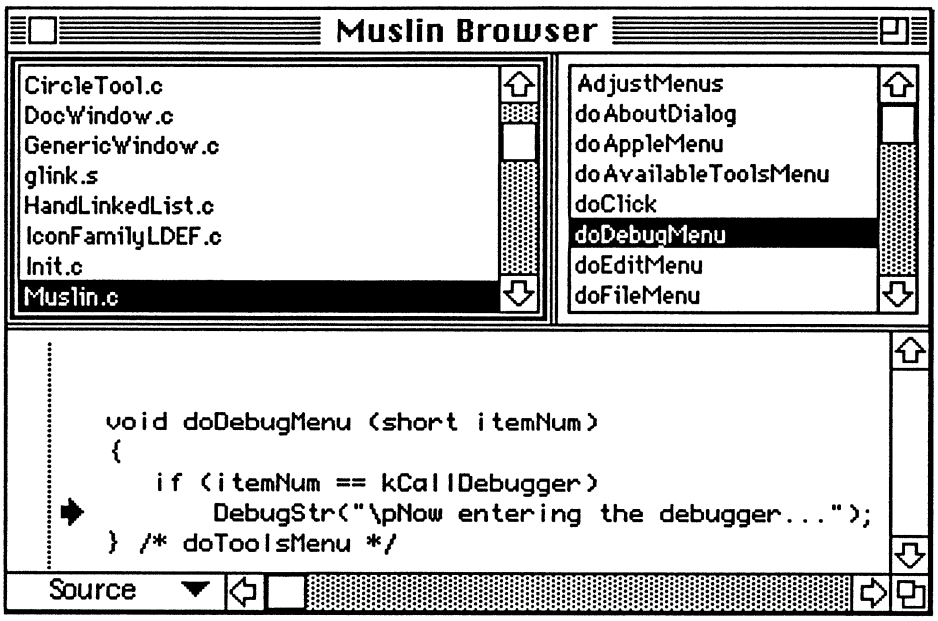
In the Muslin application, select **Debug:Invoke the debugger** now.

The target machine will appear to hang. The call to DebugStr() has given control to the debugger nub, which has no user interface. Go back to the 68K Macintosh and wait. (The host machine can take upwards of 10 seconds to recognize the first stop request in a given session. Be patient.)

After the host machine recognizes that a stop has occurred, it displays a register window.[1] Since we entered via DebugStr(), a Log window appears containing the message passed to DebugStr ("Now entering the debugger...".)

If the debugger could find the source files (or you indicated where they are), the source browser will display the source code around the breakpoint, with an arrow pointing to the current statement, as shown below.

[1] If the debugger asks where the source files are, tell it. Otherwise, you'll have to debug at the assembly level.



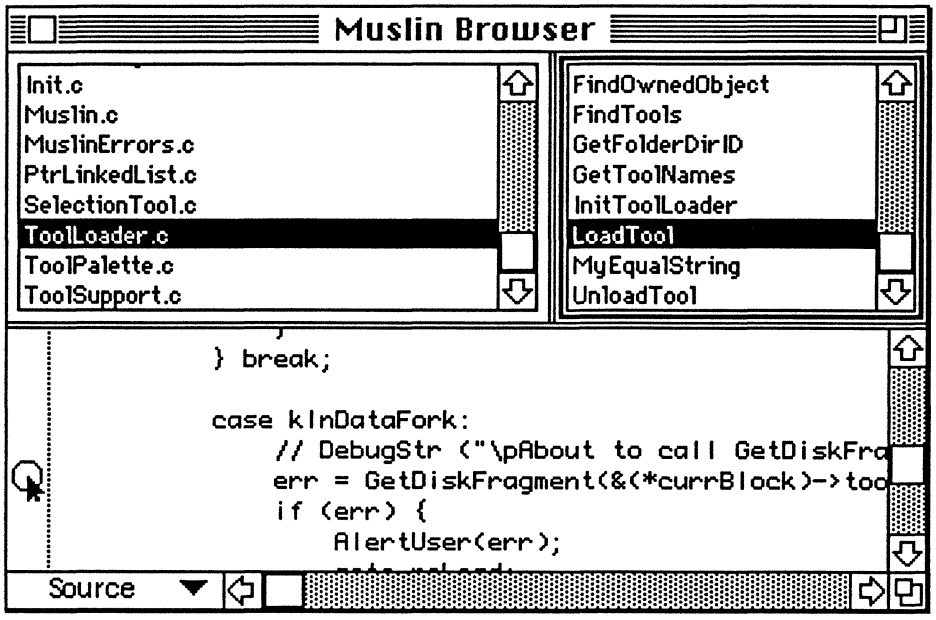
Set a breakpoint

Locate the LoadTool routine contained in ToolLoader.c, and place a breakpoint on the line containing the call to GetDiskFragment.[2] You place a breakpoint by moving the cursor into the column on the left hand side of the source window (where the cursor will turn into a small octagonal "stop" sign) and clicking.

Additional breakpoint options

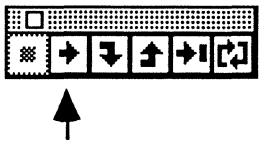
You can get other breakpoint options (including counted breakpoints, which stop only after being executed a certain number of times) by option-clicking in the left hand column.

[2] The Views:Find Source for... command provides a convenient way to locate any routine by name. alternately, you can select the source file in the upper left pane, and the function name in the upper right pane.



Run & load a tool

To continue execution, select the **Control:Run** menu command or click on the right-pointing arrow in the "control palette."



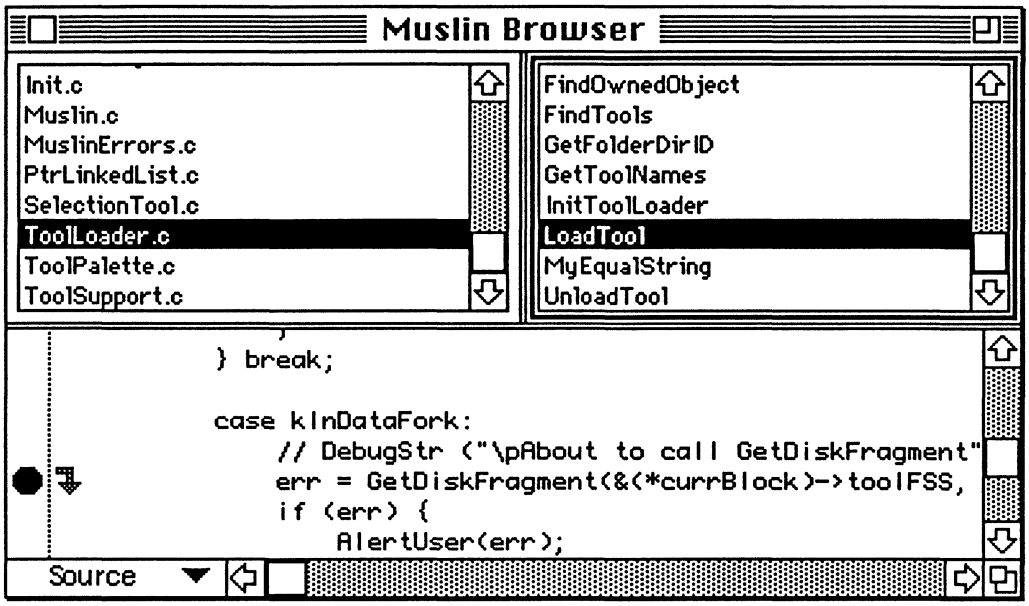
Click here

Notice that Muslin's **Debug** menu un-hilights, and control returns to the pdm. (If you forgot to unmount server volumes from the pdm, you'll have to wait several minutes for the timeouts or re-boot the machine.)

Load a tool to invoke our breakpoint

Next, select **Tools:Add Tool to palette** in Muslin, and pick a tool. This should invoke the debugger at the breakpoint you set. Again, the process of transferring control takes several seconds, at which time the PowerPC machine will appear to be hung.

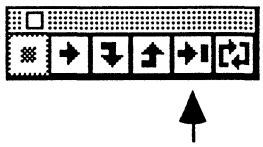
When stopped at a breakpoint, the display should look like this:



The "crooked arrow" indicates that the next "step" command will step into the specified call. (GetDiskFragment.) Since stepping into this routine would involve looking through assembly code, select the "step over" command at the right end of the palette, or set a breakpoint on the next executable line and run. (You may need to click "step over" several times to get out of the breakpoint.)

Explore some on your own

By the way, balloon help works inside R2DB. You might want to turn it on and see what some of the other buttons on the control palette do.



Debugging the plug-in tools

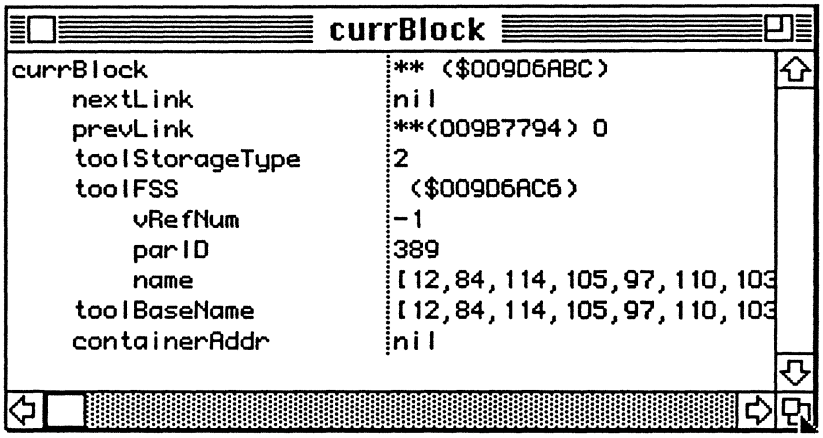
You can also load one of the SYM files for a tool (the Square tool is a good one since it is so simple) and set breakpoints in the tool's code. You might want to give this a try...

If the connection dies

The debugger sometimes loses its connection with the target machine. If that happens, just re-launch the target application, and re-invoke the debugger via DebugStr(). (You can press command-D in Muslin to do this.)

Examine some variables

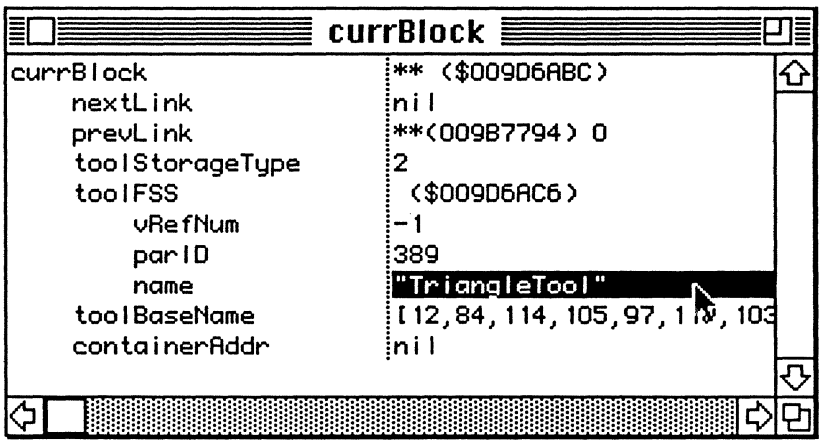
After you've tried some of the stepping options, load another tool to invoke the breakpoint in LoadTool. Now, double click on "currBlock" (in the first parameter to GetDiskFragment) and select the second command in the Evaluate menu. (It should read **Evaluate currBlock**.) This command evaluates the currently selected variable and displays its value:



The debugger also allows you to evaluate C or Pascal expressions in the "Evaluate an expression" window.

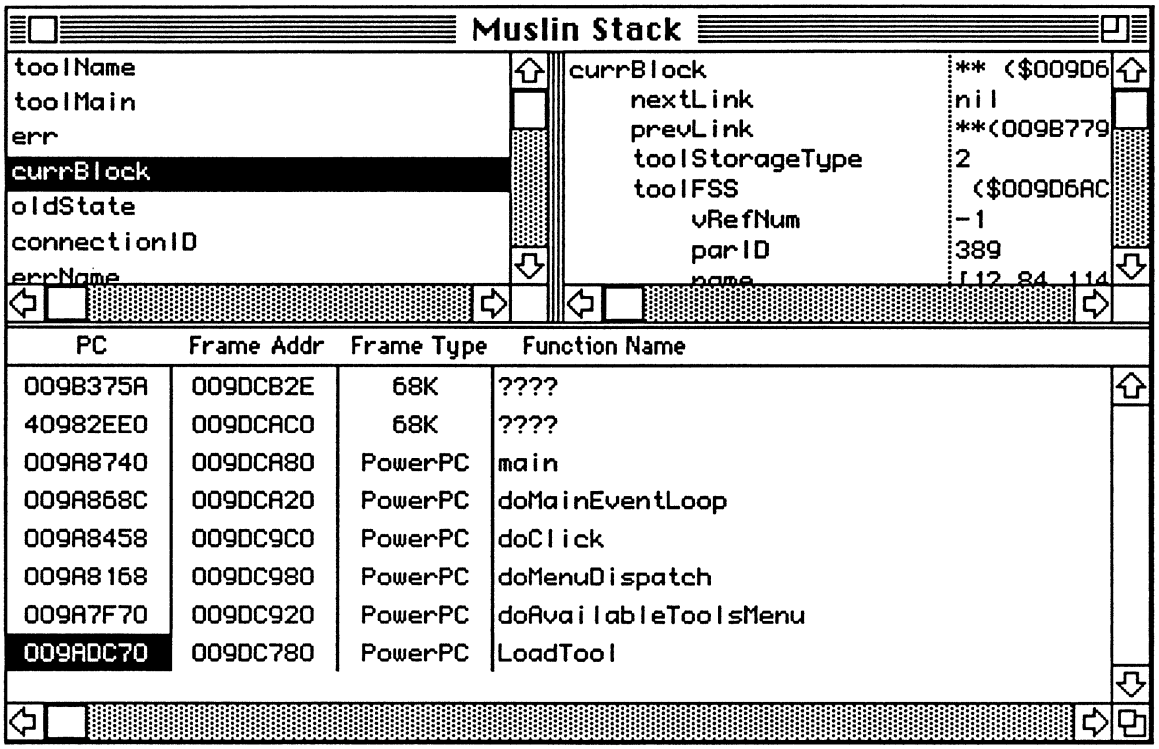
Changing the displayed format

The default format for this display places shows strings as an array of decimal bytes. To view "name" as a string, click once on the array to the right of name and wait for the line to be highlighted. (Yes, it's slow, and the line will seem to disappear. Be patient.) Then, select **Evaluate:View as Str255** and the string will appear properly.



Examine the stack frames

If you want to see the calling history up to a certain point, or want to examine several of the local variables in a routine, **Views:Show Stack Crawl** will display the following 3-part browser:



Selecting a stack frame in the lower pane will display the appropriate local variables in the upper left pane. Double-clicking one of the variables will bring up the variable examiner in the upper right pane, just as **Evaluate...** does.

Global variables can be viewed in a similar fashion by selecting **Views :Show Globals"**.

Examine disassembled code

You can view the assembly-language code for any part of the source in two different ways: 1) select Views:Show Instructions which displays the instructions beginning with the current program counter, or 2) click on the pop-up menu in the lower left corner of the source browser, and select "source" . If you have highlighted some source code, the second method will hilight the corresponding part of the disassembly.

Examining 68K code

R2DB can also disassemble 68K code, but that's the extent of its 68K abilities. You can't set breakpoints or examine variables in 68K code.

Comparing assembly & source

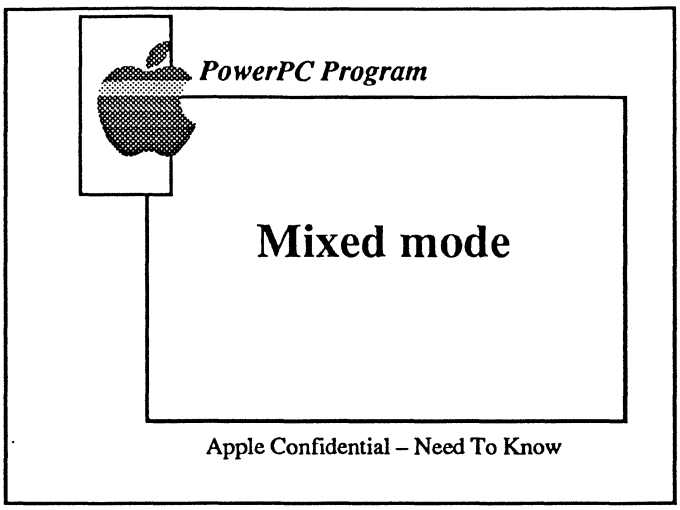
One useful technique involves duplicating a window and displaying source code in one and the corresponding disassembly in the other. You can do this by holding down the option key and dragging on a source browser window. This will cause a copy of the window to be made which you can then switch to source or assembly mode using the pop-up in the lower left corner.

A Quick Escape

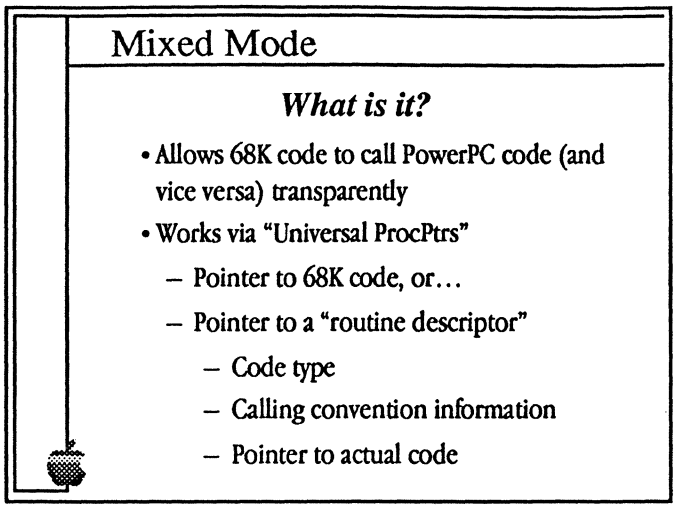
If your program crashes and you want a quick way out, selecting **Control:Enter MacsBug** will drop the target machine into MacsBug, where an ES (Exit to shell) command will kill the current application.

Further explorations

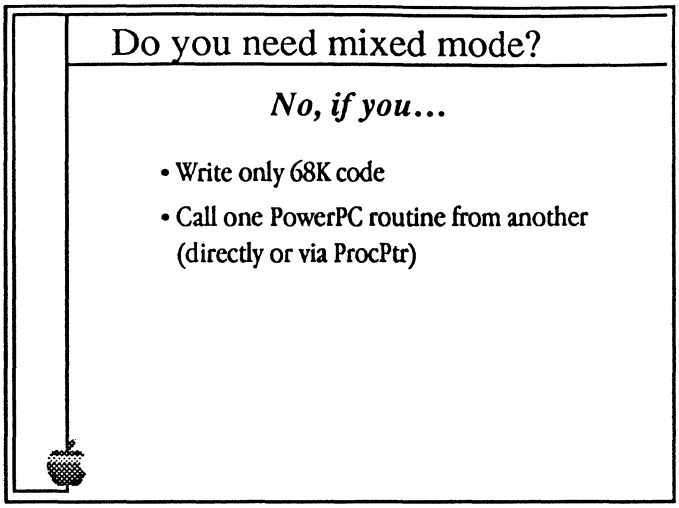
Take a look at the Memory, Registers, and FPU registers windows, all accessible from the Views menu. Try out the balloon help, and feel free to experiment!


PowerPC Program
Mixed mode
Apple Confidential – Need To Know

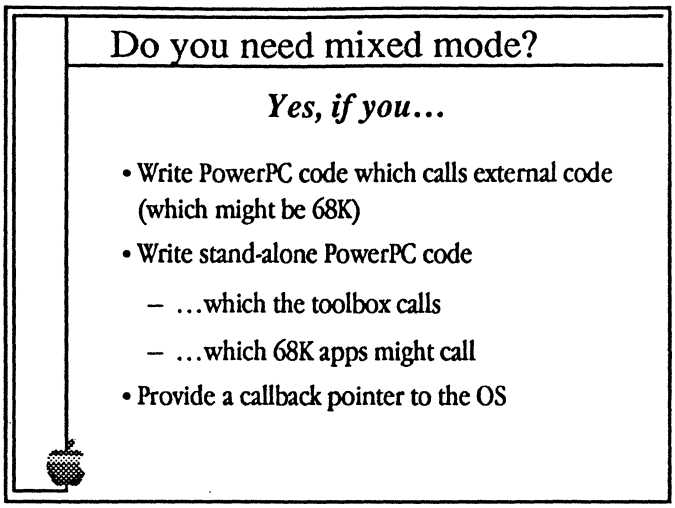
Mixed mode


Mixed Mode
What is it?
• Allows 68K code to call PowerPC code (and vice versa) transparently
• Works via "Universal ProcPtrs"
– Pointer to 68K code, or...
– Pointer to a "routine descriptor"
– Code type
– Calling convention information
– Pointer to actual code

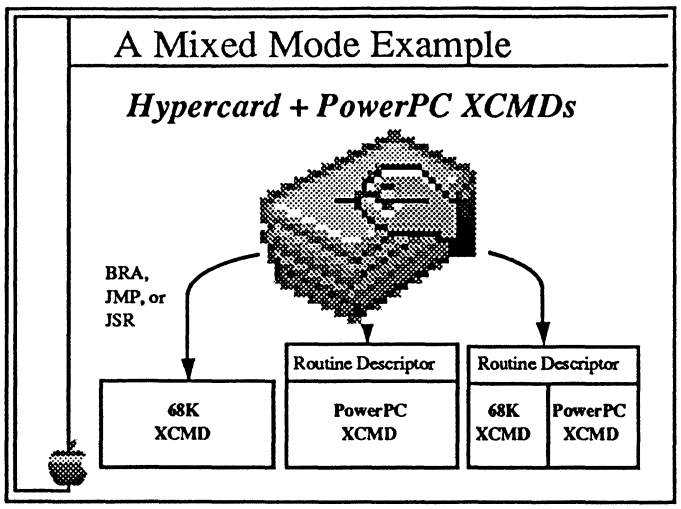
Mixed Mode



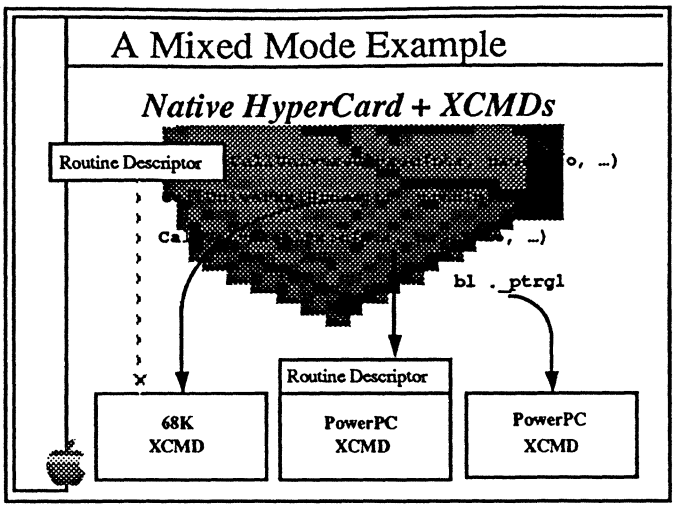
Do you need mixed mode?

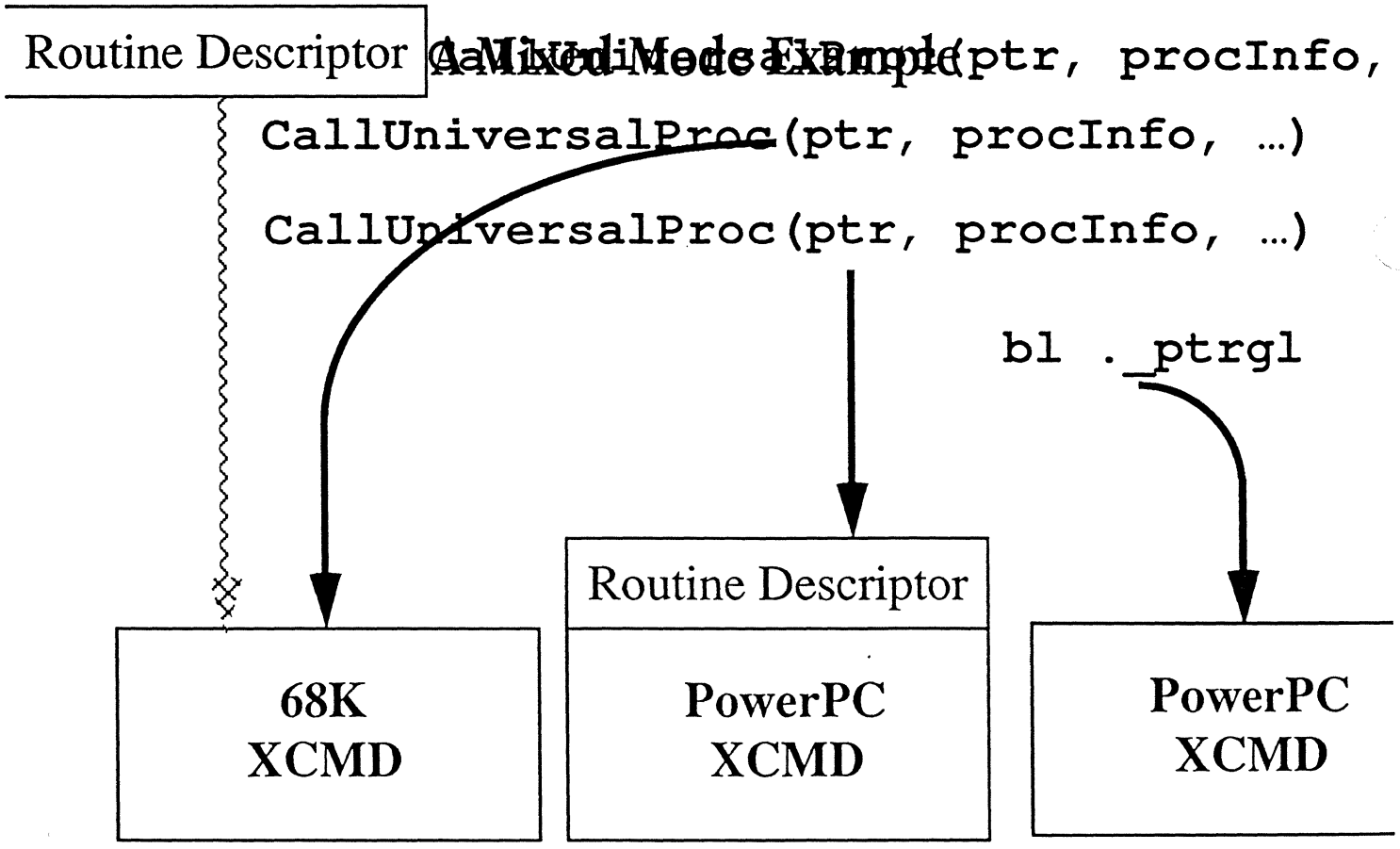


Do you need mixed mode?



A Mixed Mode Example


A Mixed Mode Example
Native HyperCard + XCMDs
Routine Descriptor
bl ._ptrgl
Routine Descriptor
68K
XCMD
PowerPC
XCMD
PowerPC
XCMD

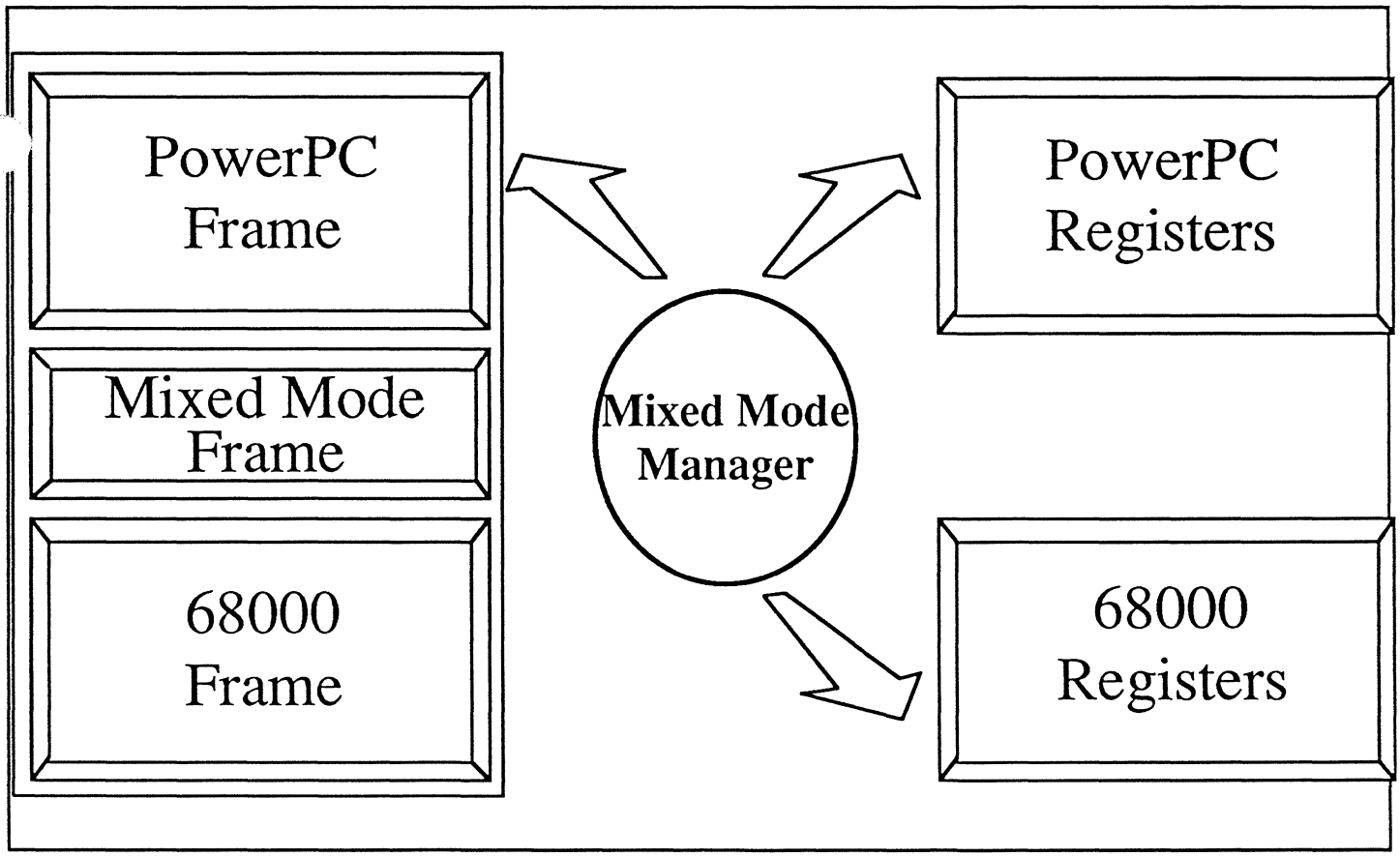

Routine Descriptor
A Mixed Mode Example
CallUniversalProc(ptr, procInfo,
CallUniversalProc(ptr, procInfo, ...)
CallUniversalProc(ptr, procInfo, ...)
bl ._ptrgl
Routine Descriptor
68K
XCMD
PowerPC
XCMD
PowerPC
XCMD

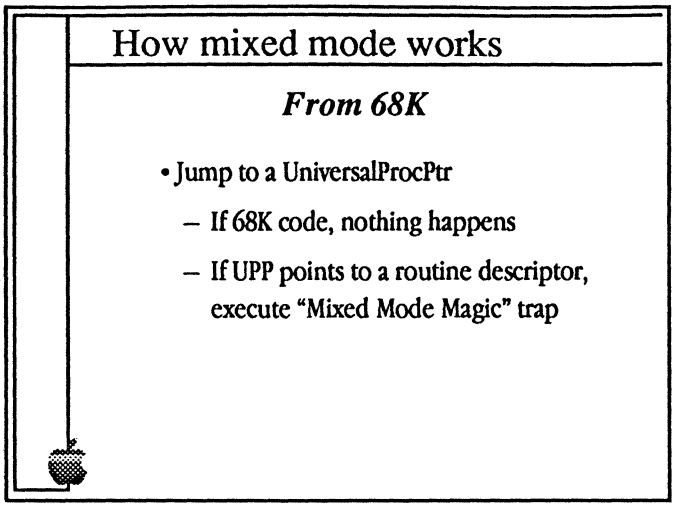
How mixed mode works

From PowerPC

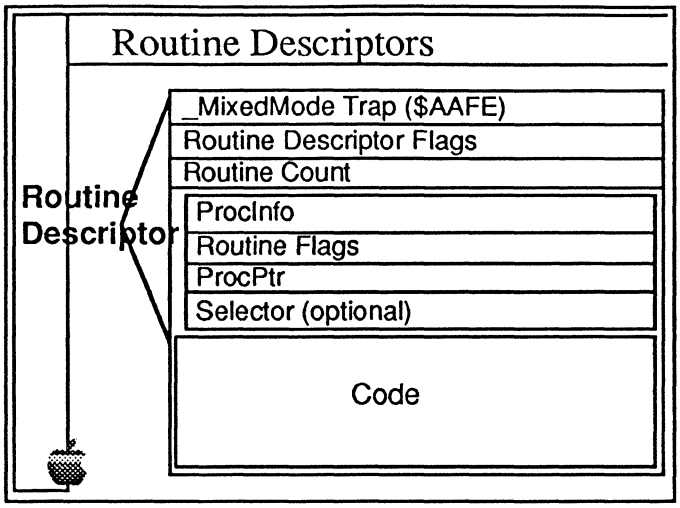
- Execute CallUniversalProc w/ UPP & ProcInfo
  - UPP == 68K pointer, or pointer to a RoutineDescriptor
- Format parameters for callee
  - 3 stack frames: original, "switch", & target
- If destination code != PowerPC, run emulator
- On return, reverse switch occurs

How mixed mode works



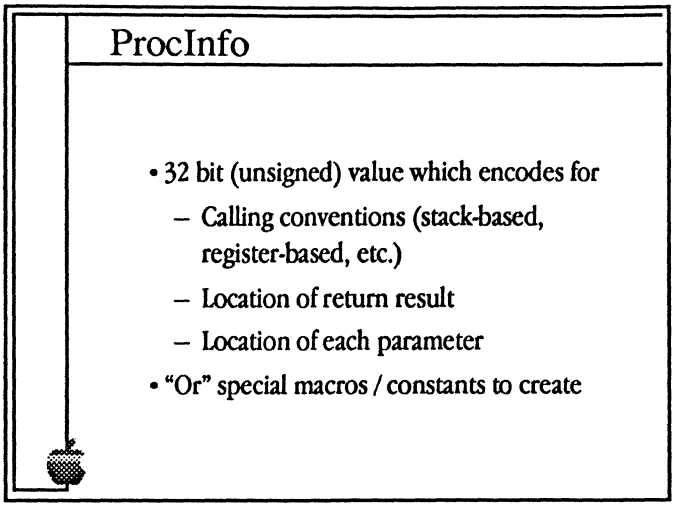


How mixed mode works



Routine Descriptors

```
struct private_RoutineDescriptor {
      long           goMixedMode;    // unused 68K instruction          0
      long           magicCookie;    // unique identifier               4
      SelectorType   selector;       // mixed mode selector             8
      short          version;        // 0 for now, increment as needed  10
      ProcInfoType   procInfo;       // calling conventions             12
      ProcPtr        customParamProc;// Procedure to convert params     16
                                     // to different types of code
                                     // Presently unused
      CodeType       executionMode;  //                                 20
      ProcPtr        procDescriptor; // Pointer to actual routine       24
      long           unused;         // must be 0                       28
};
```



ProcInfo

Calling convention types

```
kPStackBased
kCStackBased
kRegisterBased
kD0DispStackBased
kD1DispStackBased
kStackDispStackBased
kStackDispRegisterBased
kSpecialCaseProcInfoType
```

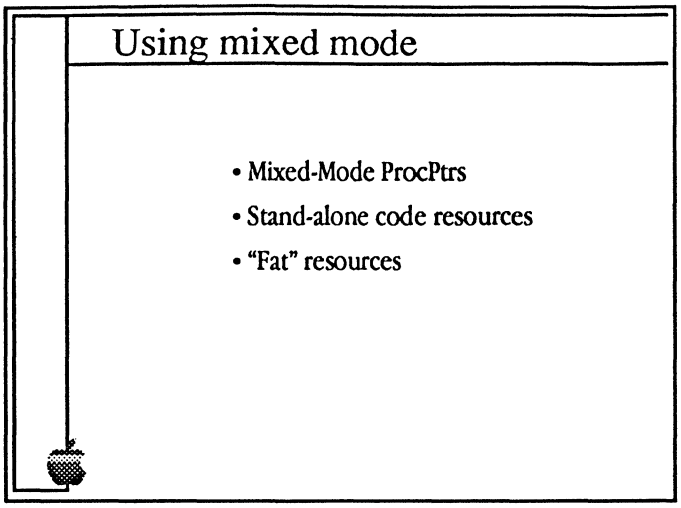
Parameter Types & Locations

```
m68KStackP[1-13][Byte,Word,Long]

m68KRegisterP[1-4][In,Out,InOut]
m68KRegisterP[1-4][Byte,Word,Long]
m68KRegisterP[1-4][A0,A1,D0,D1]

m68KStackSelector[Byte,Word,Long]
m68KRegSelector[Byte,Word,Long]

kNoReturnValue
k68K[Byte,Word,Long]Returned
```

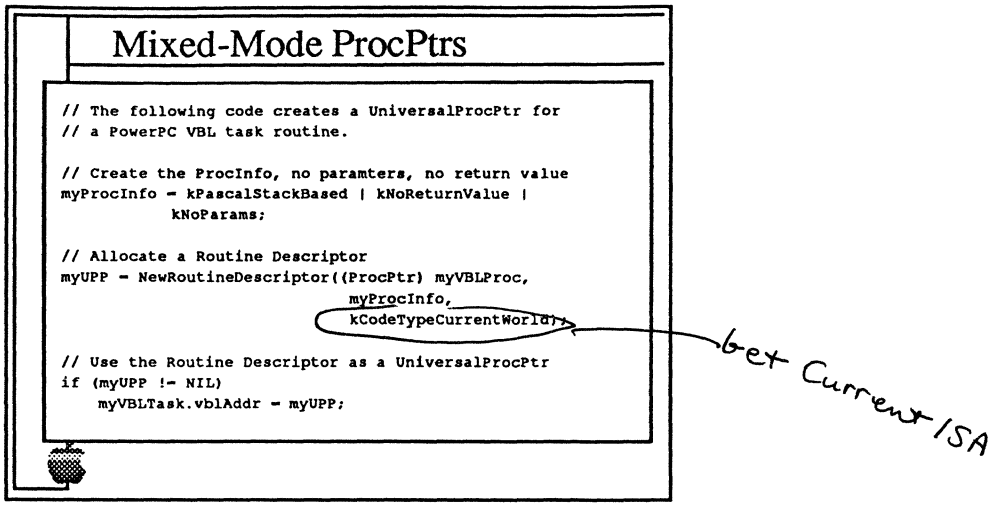
Special cases

```
kSpecialCaseHighHook
kSpecialCaseCaretHook
kSpecialCaseEOLHook
kSpecialCaseWidthHook
kSpecialCaseNWidthHook
kSpecialCaseTextWidthHook
kSpecialCaseDrawHook
kSpecialCaseHitTestHook
kSpecialCaseTEFindWord
kSpecialCaseADBRoutines
kSpecialCaseProtocolHandler
kSpecialCaseSocketListener
```

```
Explanation of special cases:
*****************************
1 = C calling conventions, Rect on stack, pointer in A3, no return value
2 = Register-based; inputs in D0, A3, A4; output is Z flag of
      status register (see VI-15-26)
3 = Register-based; inputs in D0, D1, A0, A3, A4; output in D1 (see VI-15-27)
4 = Register-based; inputs in D0, D1, D2, A0, A2, A3, A4; output in D1
      (see VI-15-27)
5 = Register-based; inputs in D0, D1, A0, A3, A4; output in D1 (see VI-15-28)
6 = Register-based; inputs in D0, D1, A0, A3, A4; no output (see VI-15-28)
7 = Register-based; inputs in D0, D1, D21, A0, A3, A4; outputs in
      D0, D1, D2 (See VI-15-29)
8 = Register-based; inputs in D0, D2, A3, A4; outputs in D0, D1 (see VI-15-30)
9 = Register-based; inputs in A0, A1, A2, D0; no outputs (see V-371)
A = Register-based; inputs in A0, A1, A2, A3, A4, D1.w; output in Z (see II-326)
B = Register-based; inputs in A0, A1, A2, A3, A4, D0.b, D1.w; output in Z
      (see II-329)
```

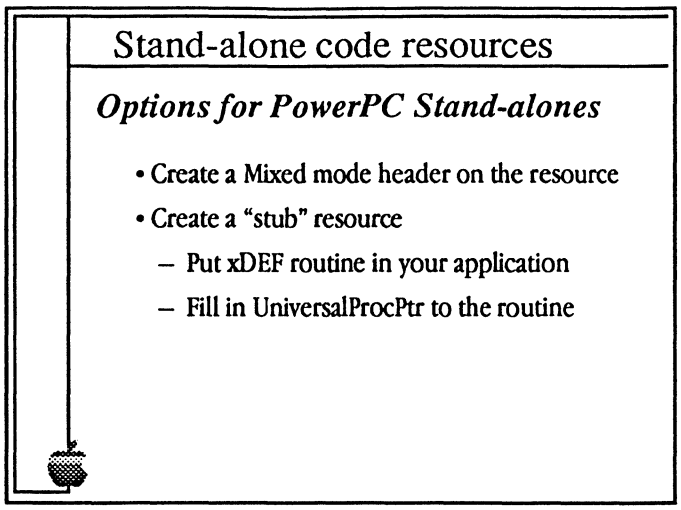


Using mixed mode

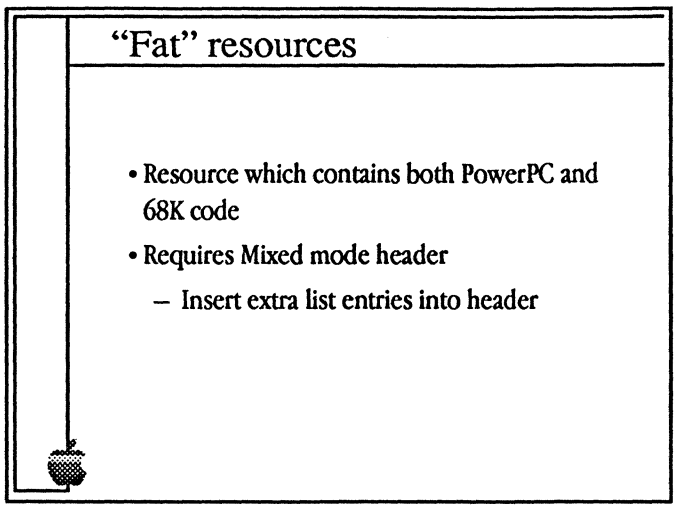


Mixed-Mode ProcPtrs

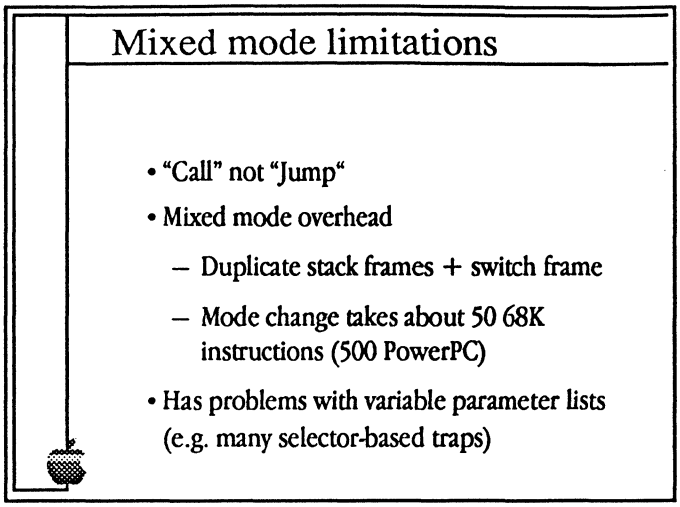
- Code types:
  - unknown
  - Current World
  - 68K
  - PowerPC



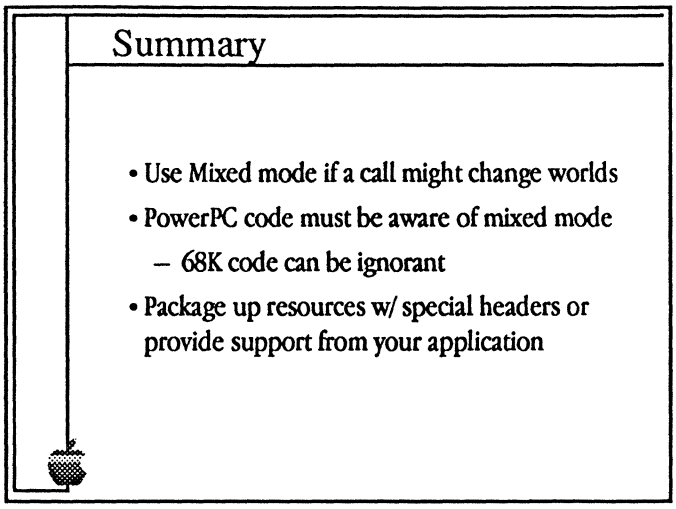
Stand-alone code resources



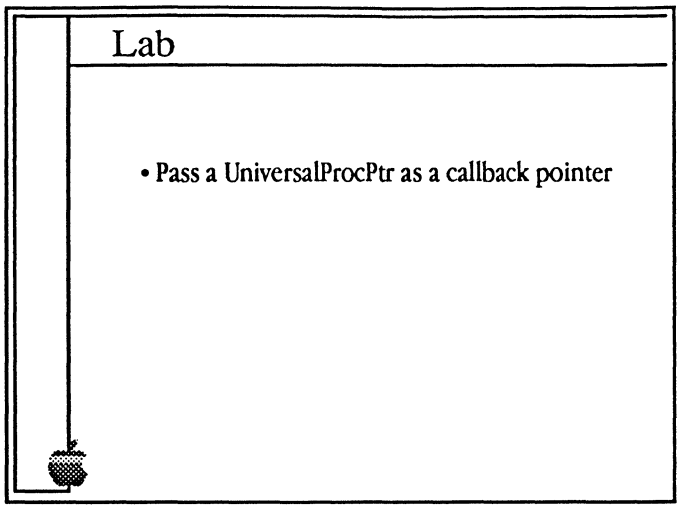
"Fat" resources



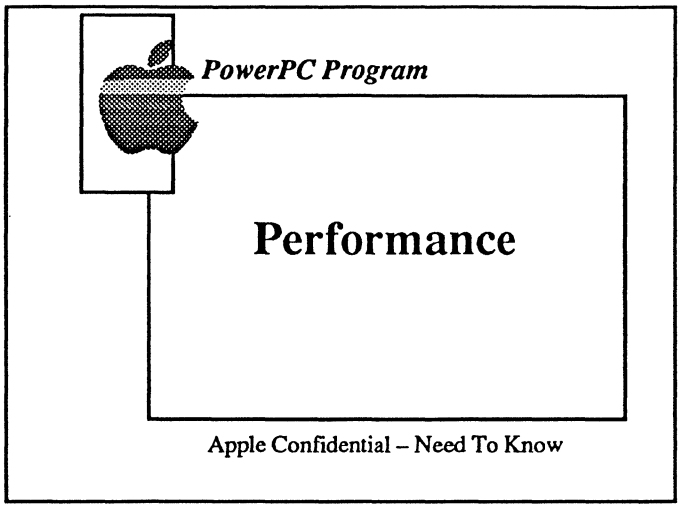
Mixed mode limitations


Summary
• Use Mixed mode if a call might change worlds
• PowerPC code must be aware of mixed mode
– 68K code can be ignorant
• Package up resources w/ special headers or provide support from your application

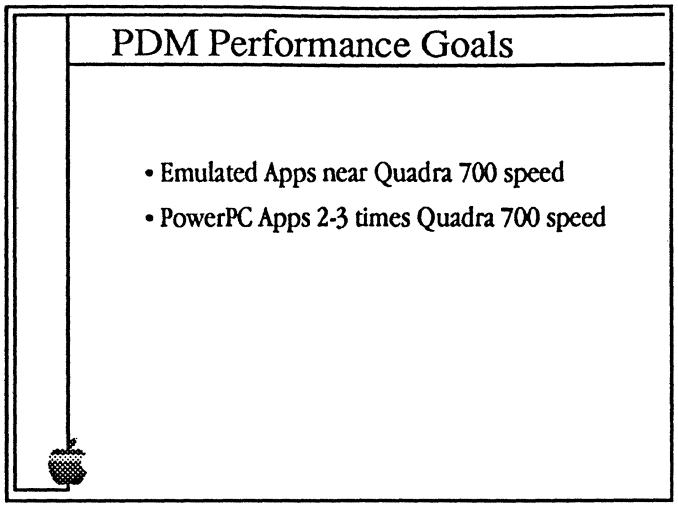
Summary



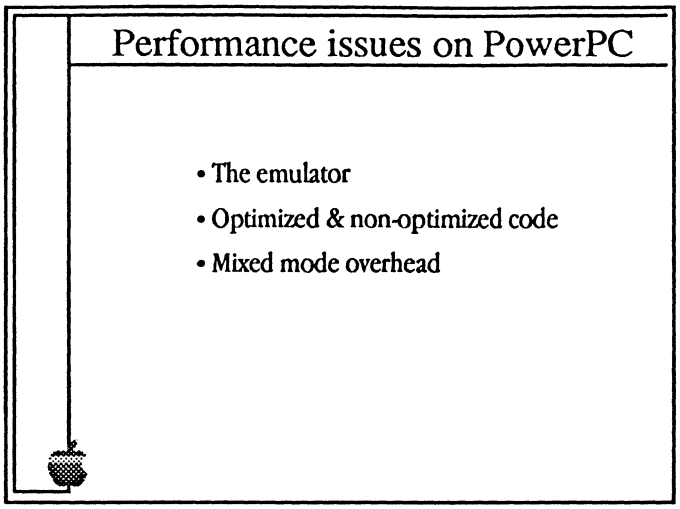
Lab



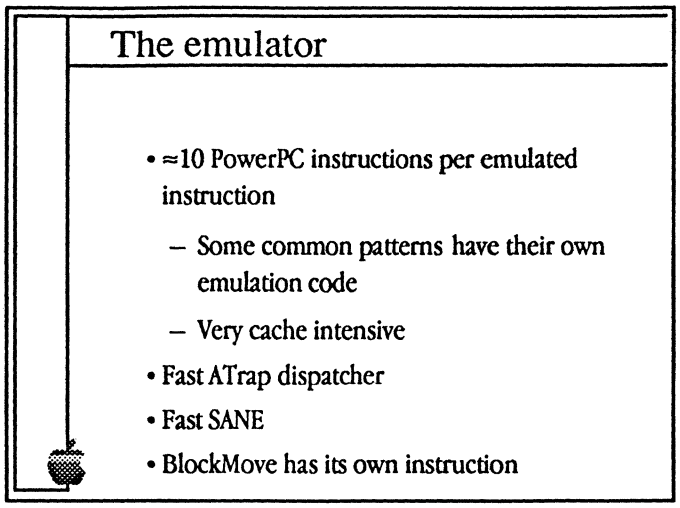
Performance



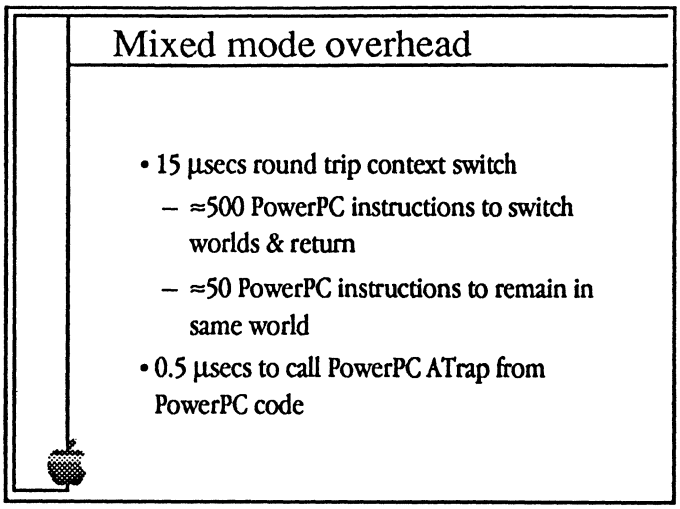
PDM Performance Goals



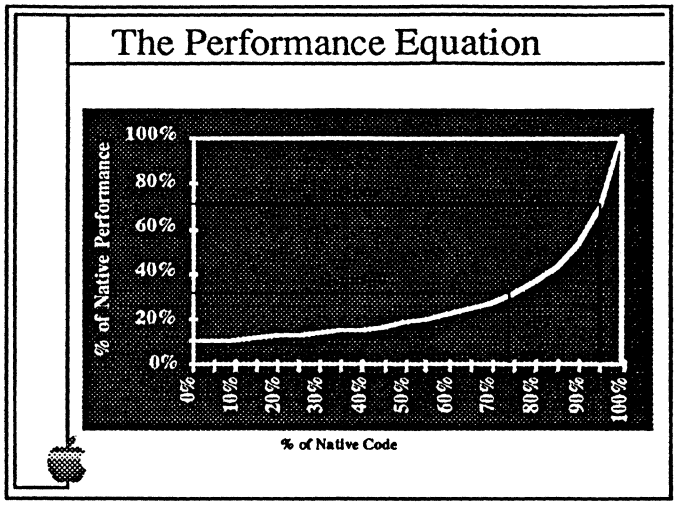
Performance issues on PowerPC



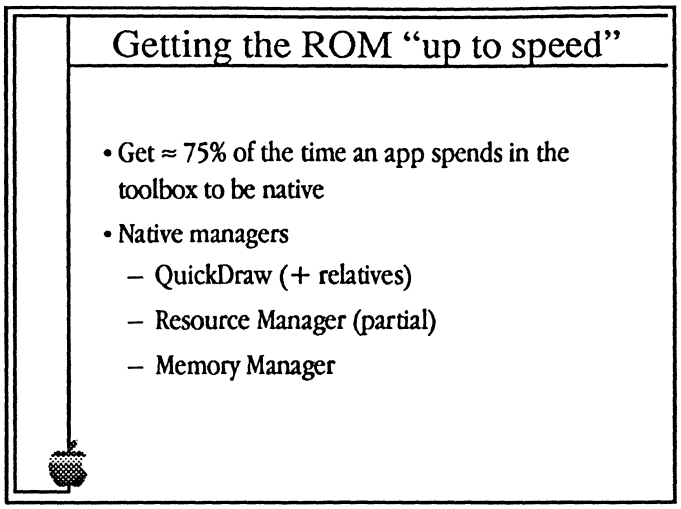
The emulator



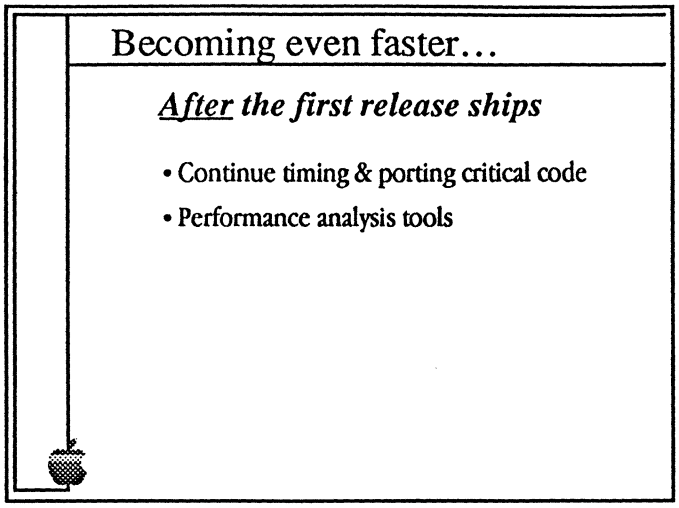
Mixed mode overhead



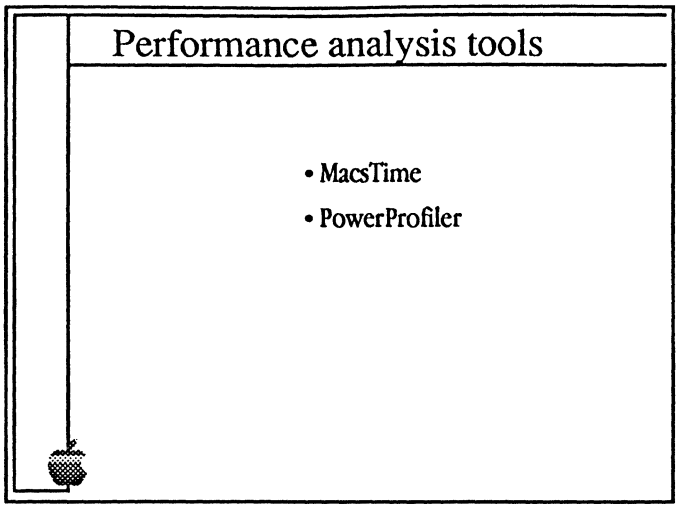
The Performance Equation



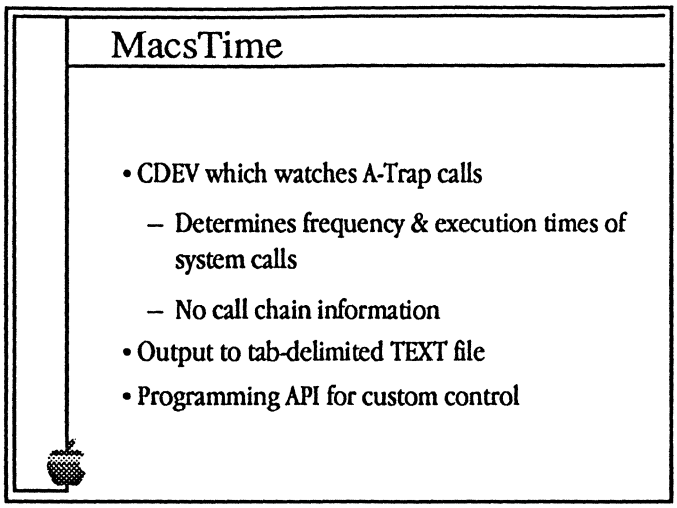
Getting the ROM "up to speed"



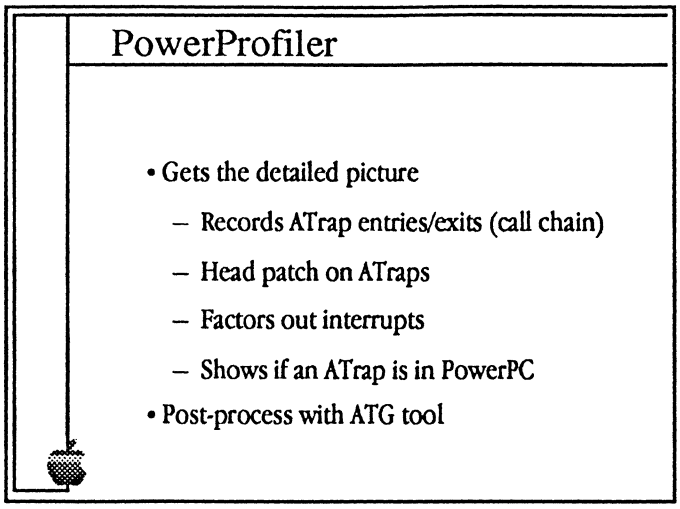
Becoming even faster...



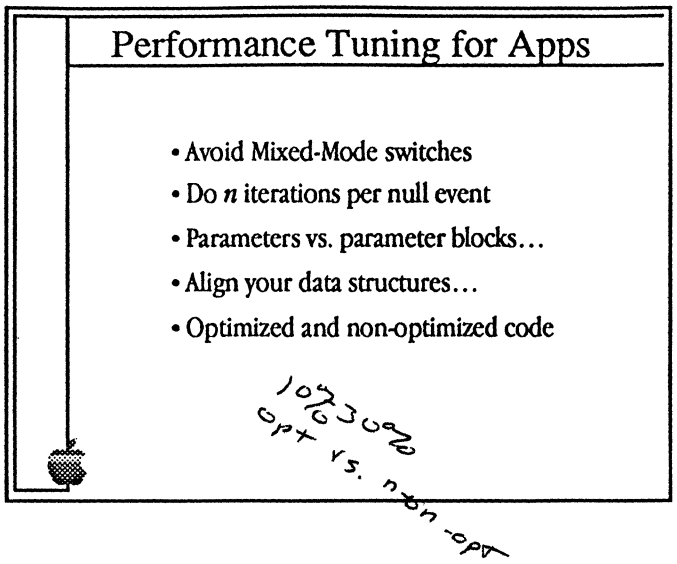
Performance analysis tools


MacsTime
• CDEV which watches A-Trap calls
– Determines frequency & execution times of system calls
– No call chain information
• Output to tab-delimited TEXT file
• Programming API for custom control

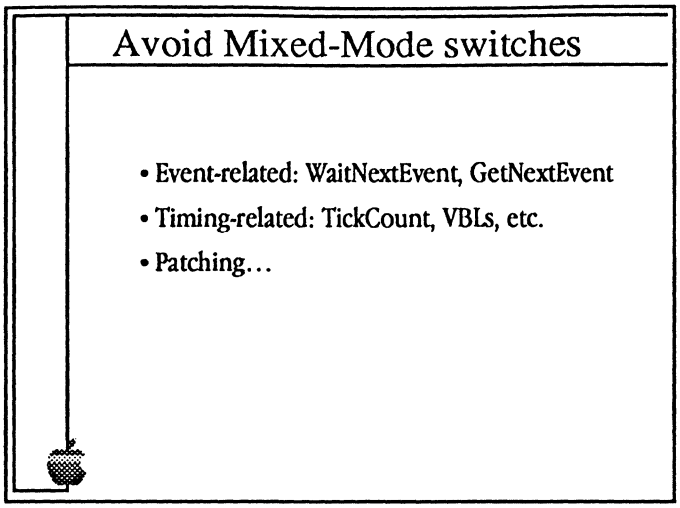
MacsTime



PowerProfiler

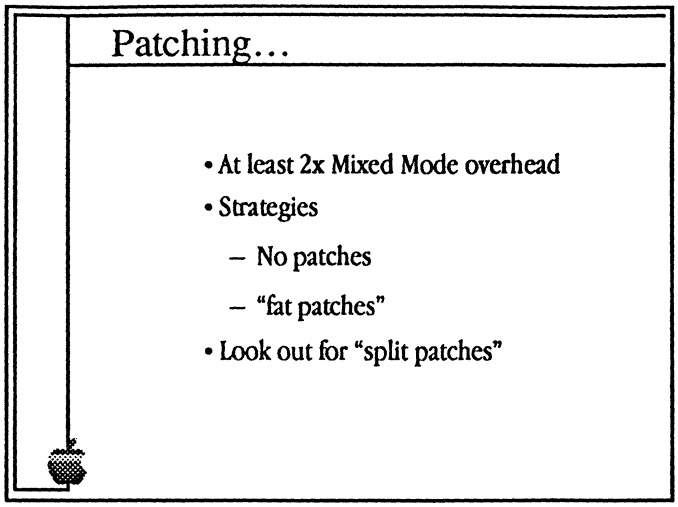


Performance Tuning for Apps

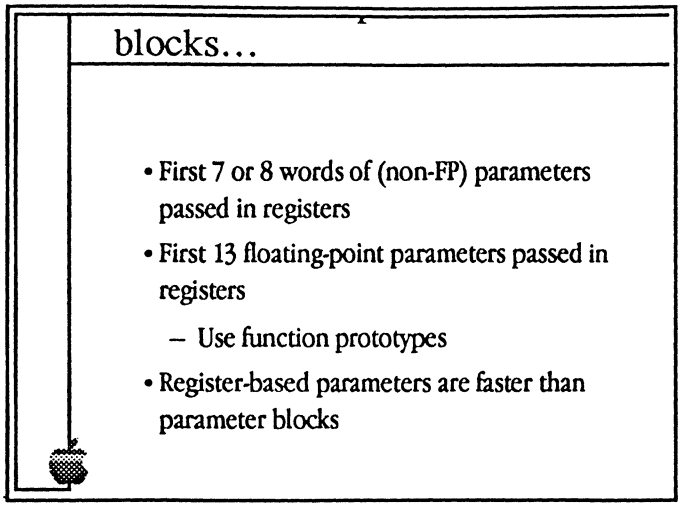


Release notes list native vs emulated traps

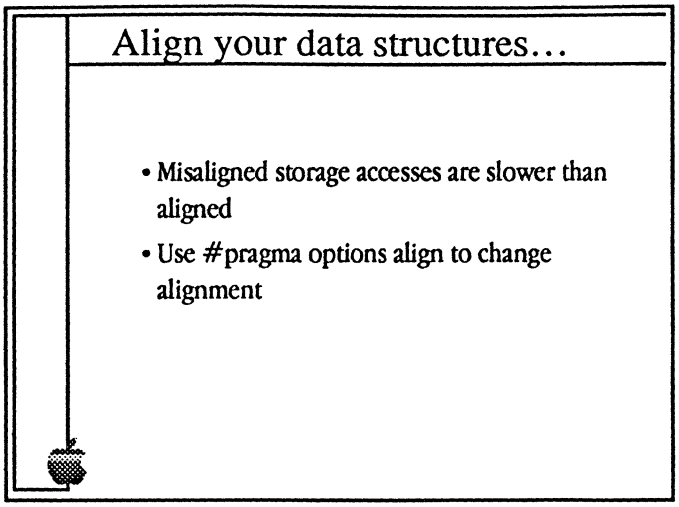
Avoid Mixed-Mode switches


Patching...
• At least 2x Mixed Mode overhead
• Strategies
– No patches
– "fat patches"
• Look out for "split patches"

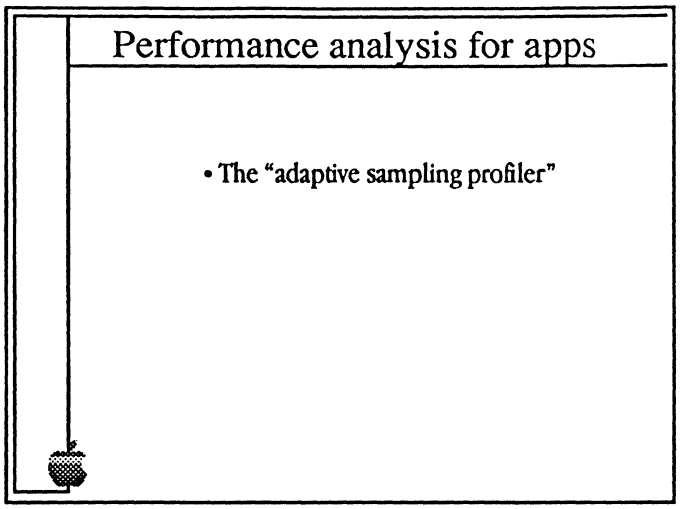
Patching...



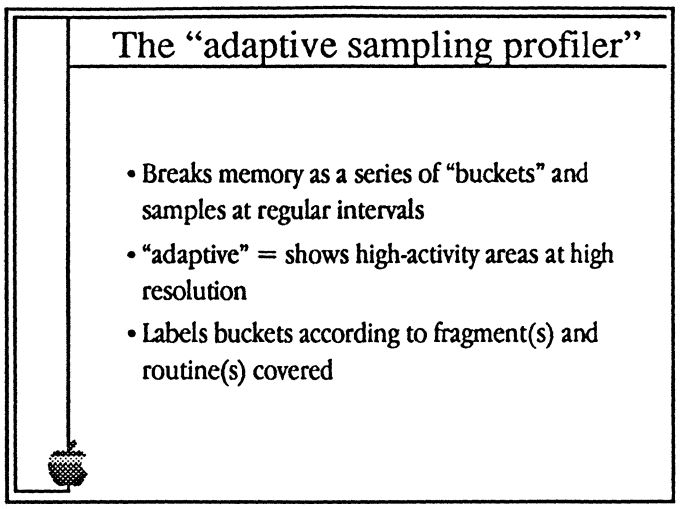
Parameters vs. parameter blocks...



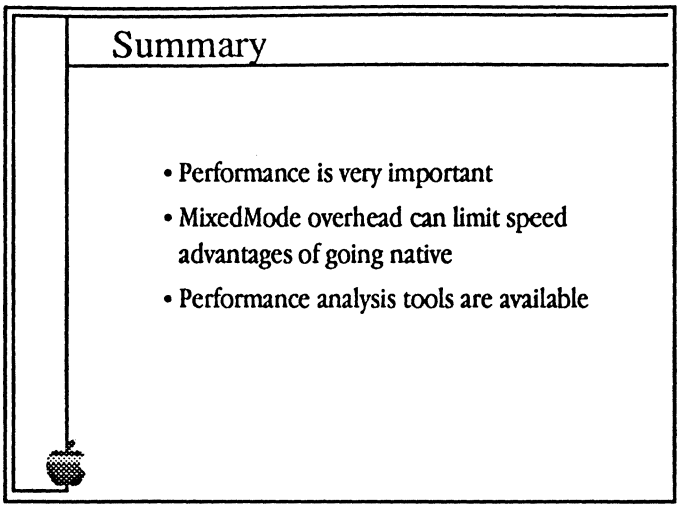
Align your data structures...



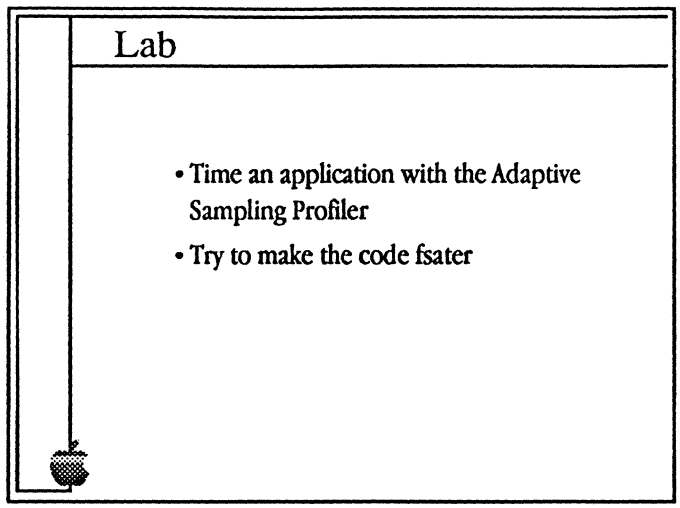
Performance analysis for apps



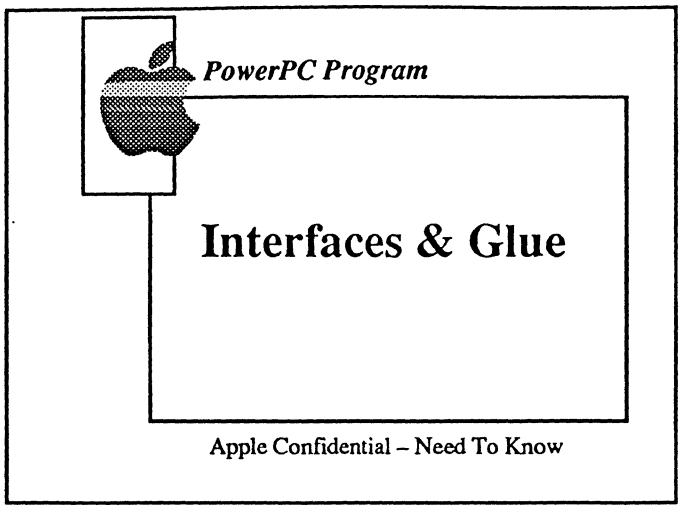
The "adaptive sampling profiler"


Summary
• Performance is very important
• MixedMode overhead can limit speed advantages of going native
• Performance analysis tools are available

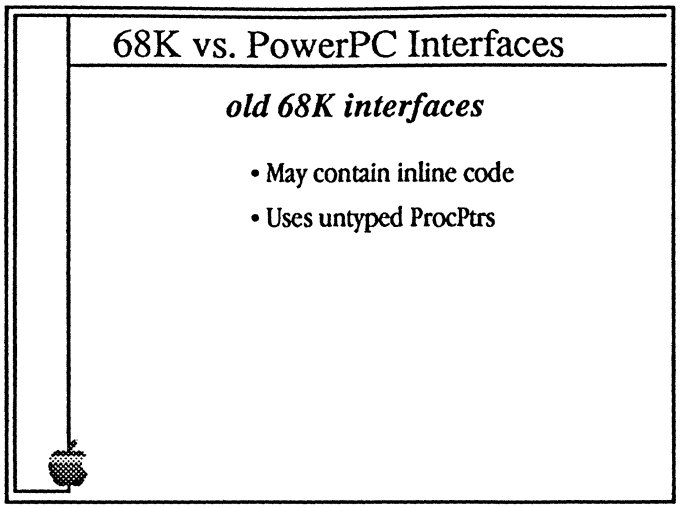
Summary


Lab
• Time an application with the Adaptive Sampling Profiler
• Try to make the code fsater

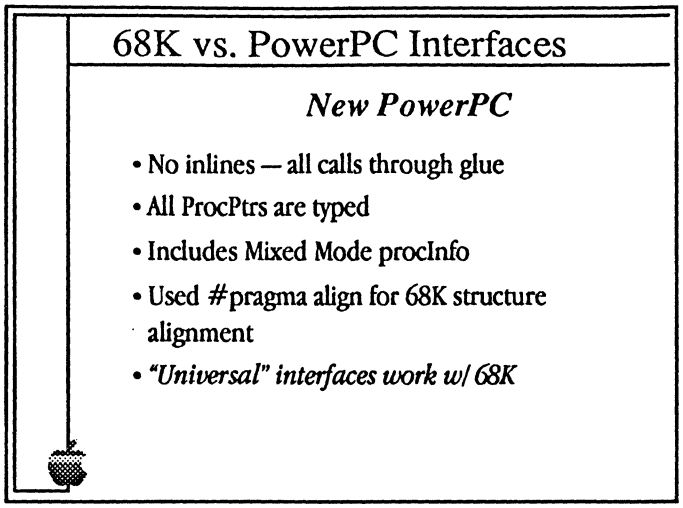
Lab



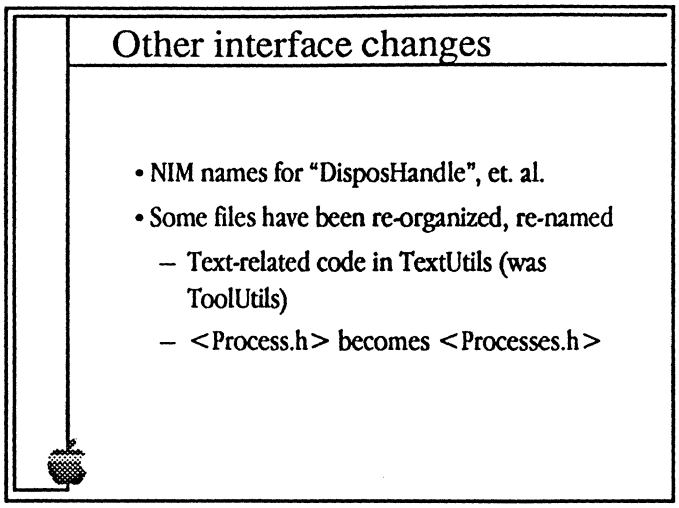
Interfaces & Glue



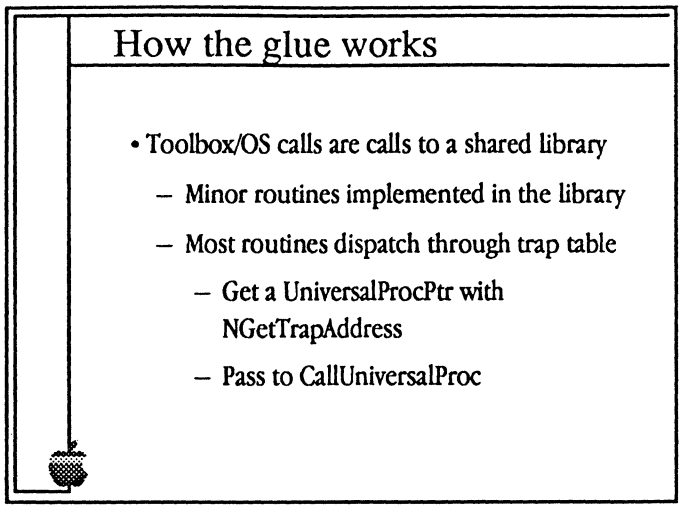
68K vs. PowerPC Interfaces



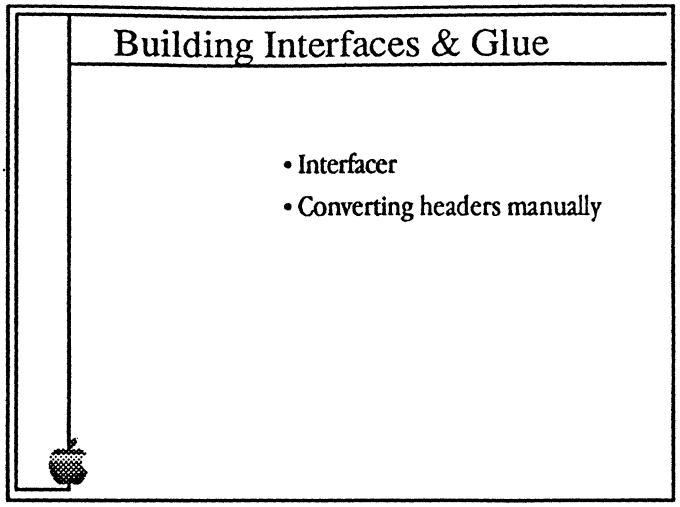
68K vs. PowerPC Interfaces



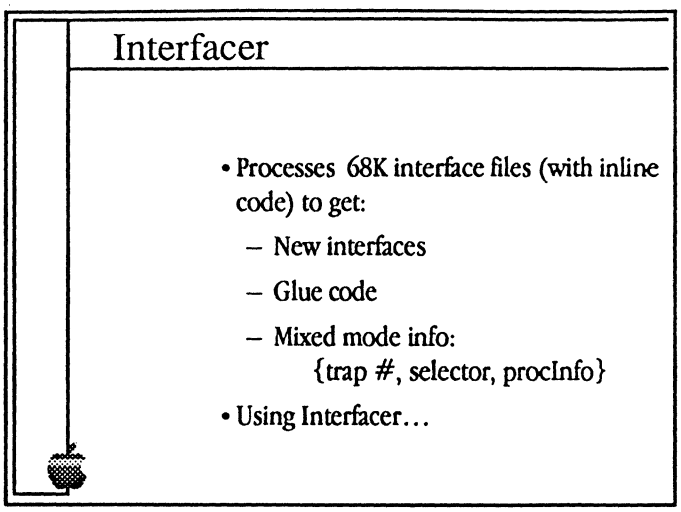
Other interface changes



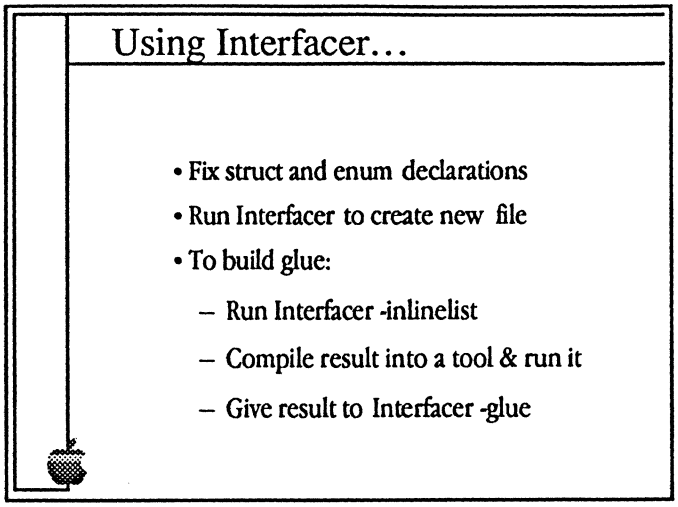
How the glue works



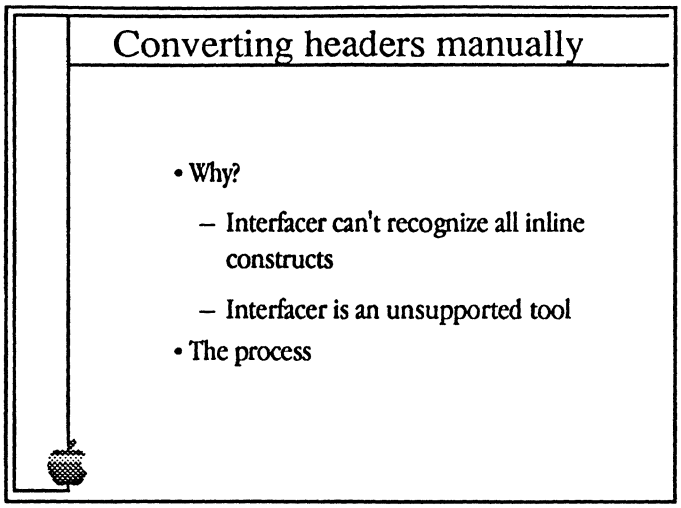
Building Interfaces & Glue


Interfacer
• Processes 68K interface files (with inline code) to get:
– New interfaces
– Glue code
– Mixed mode info:
{trap #, selector, procInfo}
• Using Interfacer...

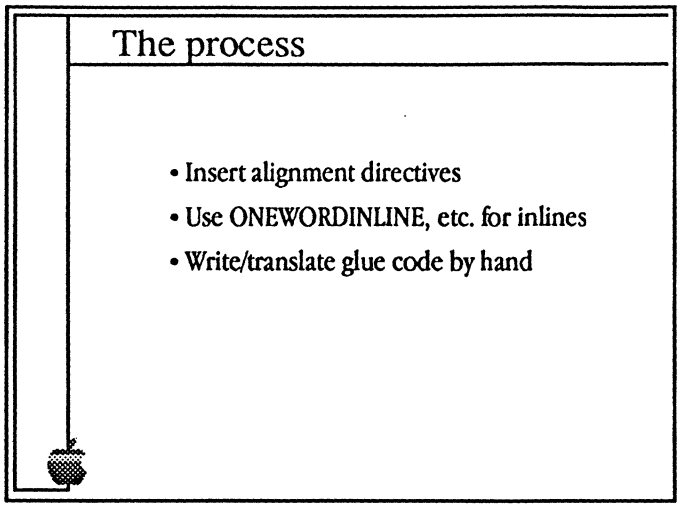
Interfacer



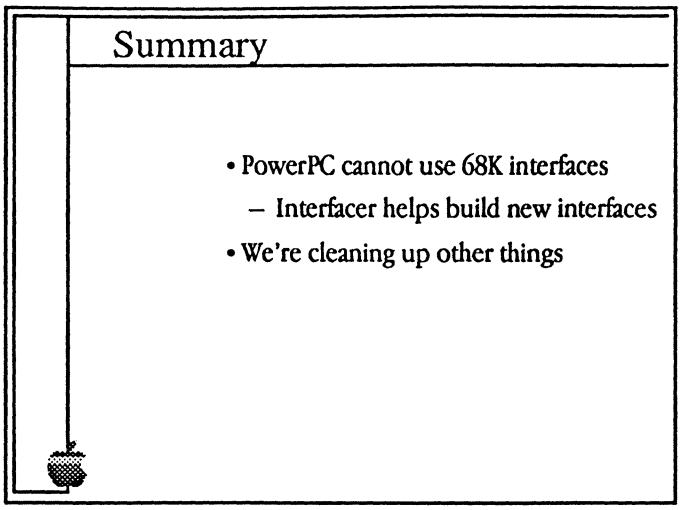
Using Interfacer...



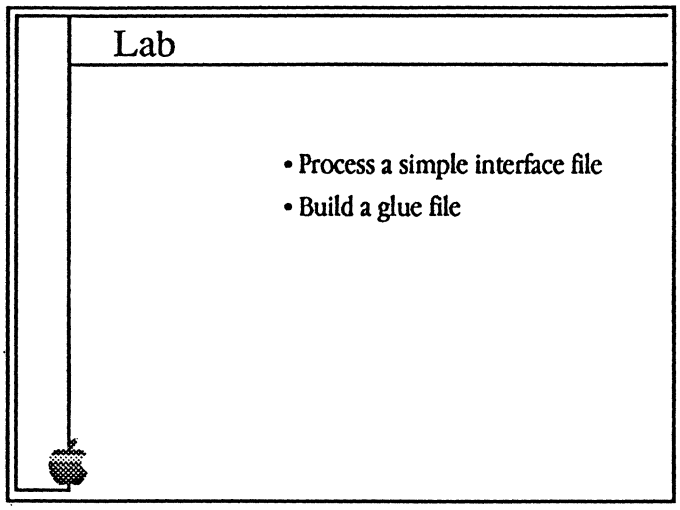
Converting headers manually


The process
• Insert alignment directives
• Use ONEWORDINLINE, etc. for inlines
• Write/translate glue code by hand

The process


Summary
• PowerPC cannot use 68K interfaces
– Interfacer helps build new interfaces
• We're cleaning up other things

Summary



Lab

PowerProfiler

Rev 1.0
Jim Gochee
2/18/93

Introduction

The PowerProfiler is a system profiler and timing tool. It's main objective is to time and show the calling chain of system ATraps. There already exist some excellent ATrap timing tools, however they currently use the ATrap dispatcher to record trap calls. The PowerProfiler head patches every ATrap to be profiled so that it is 100% guaranteed of intercepting an ATrap call. This is necessary for Quickdraw which normally jumps through the ATrap vector instead of paying the overhead of invoking the A-line exception. Also, native code on PowerPC machines uses MixedMode to make toolbox calls and MixedMode jumps through the vector directly.

Installation

The PowerProfiler comes in three pieces. The first is ZProfileInit, which as its name suggests is an init that runs very late in the boot process. It is responsible for patching the ATraps to be profiled. It also allocates a block of memory in the system heap for the profile data. There are some MacsBug templates and macros in ZProfileInit which should be copied into the Debugger Prefs file in your system folder. If you don't know what a debugger prefs file is, or you haven't used MacsBug then you probably should not be using the PowerProfiler. The resources you will need to copy from ZProfileInit are 'mxbm' and 'mxwt'. When this is done, drag ZProfileInit into you system folder (extensions folder) and restart to install the profiler.

The other two things you will need are SnarfResults and TrapSum. SnarfResults is an application that writes the current profiling data to an MPW readable text file. The data from the current profile is stored in a block in the system heap and is accessed through a data structure pointed to by location $100 in memory. TrapSum is an MPW tool that can post-process the file produced by SnarfResults and give summary information for each trap in the profile.

ZProfileInit

At init time, ZProfileInit does several things. First, it sets up a data structure pointed to by location $100 in memory. This data structure contains information about the current profile setup, the most interesting to the user being the 'state' of the profile and the remaining buffer space. By default, the profiler is de-activated (state = 0) and no profiling information is recorded. The second thing the init does is patch a list of ATraps described in the 'PROF' resource of the file. Then it patches the interrupt vectors.

To display the profiler's data structure, drop into MacsBug and type 'table'. The key fields are 'state', and 'ElementCount', which shows how many ATrap in/out entries have been logged. Since the profiler is turned off by default, both of these numbers should be 0. Note that the PowerProfiler is nevertheless plugged into the ATrap table and is being called through at each trap instance. This is important because some ATraps don't like to be profiled. The profiler assumes it can replace the caller's address on the stack with it's own internal address. In this way, the ATrap will return to the profiler. If the ATrap discards the return address, then the profiler's internal data structures will be corrupted and the machine will bomb. If this happens, type 'table' from MacsBug and look at the 'LastTrapReturned' or 'LastTrapInFrom' values. The trap(s) listed here are probably the culprit. Use ResEdit to zero out the bad trap from the 'PROF' resource of ZProfileInit.

To start recording, drop into MacsBug and type 'start'. This sets the 'state' to 1 and triggers the ATrap patches to save their data. The data is saved as an array of ATrap entries and exits, so the amount of information you can record is directly related to the size of the profile buffer. The

default buffer size is 300k, which is enough for about 10,000 traps to be profiled. This may seem like a lot, but it is realistically 3-10 seconds worth of information. To increase the size of the buffer, use ResEdit to change the hex value in the 'sysz' resource. The actual size of the buffer will be the value of 'sysz' minus around 20k.

To stop record, drop back into MacsBug and type 'stop'. You can then type 'table' to see how many ATraps were recorded ('ElementCount').

SnarfResults

To recover the stored profile data, launch the SnarfResults application. A file called 'SnarfResults.out' will be created in the same directory as SnarfResults. It contains the profiling output in an MPW readable text file. Here's a sample of what's generated:

```
 _DrawText (from 0x00B246EA, ApplZone 0x008410C4) 5 calls, µsecs: 2762
•       _StdText (from 0x408712A2, ApplZone 0x008410C4) 4 calls, µsecs: 2675
•               _StdTxMeas (from 0x00B2E48C, ApplZone 0x008410C4) 1 calls, µsecs: 408
•                       _FMSwapFont (from 0x00B2E2AC, ApplZone 0x008410C4) 0 calls, µsecs: 124
                _QDExtensions (from 0x00B2E48C, ApplZone 0x008410C4) 0 calls, µsecs: 39
** Interrupt
 _GetPalette (from 0x00859058, ApplZone 0x008410C4) 0 calls, µsecs: 34
 _RGBForeColor (from 0x00859076, ApplZone 0x008410C4) 1 calls, µsecs: 131
        _Color2Index (from 0x408877BC, ApplZone 0x008410C4) 0 calls, µsecs: 77
```

Each line describes an ATrap that was called, including where it was called from, what the ApplZone was, how many traps were in turn called, and the total trap time not factoring out the sub calls that were made. The indentation of the trap shows how the calling sequence proceeded. For instance, DrawText called StdText, which in turn called StdTxMeas and QDExtensions. StdTxMeas called FMSwapFont.

The bullet in front means that a trap was patched by a PowerPC version. Notice too that an interrupt occurred and was logged.

TrapSum

TrapSum is the final component of the Profiler. It runs as an MPW tool and takes as a parameter the name of a file to summarize. You can either summarize an entire SnarfResults.out file, or cut and paste to get a smaller subset. TrapSum dumps the summary to stdout.

MacsTime, the Second Version

User's Guide & Programmer's Reference

1.0 Introduction

This document describes the MacsTime trap timing and profiling tool at both the user level and the programming level. This document is broken into two main sections. The first section is the documentation for the tool at the user level. The second section contains the information needed to program the tool for specialized tasks.

1.1 What MacsTime Is

MacsTime is a tool in the form of an INIT/CDEV which profiles the execution of traps by the system and applications in a particular test case. It provides the count of trap execution (selector based traps will be timed individually by selector), the total time spent in the trap, the minimum and maximum times and the sum of squares of the trap times.

The timing process can be started or stopped on any of the following criteria: unconditionally via a button press, on some time delta from the present, on the Nth execution of some trap or on a sequence of 1 to 4 traps. Once a test case is complete, the user may recover the generated data using the control panel or may write a tool to return the data in whatever format is required.

The division of labor runs as follows: The INIT takes timing commands through its command procedure, performs the actual trap timing and returns pointers to its database. The CDEV acts as a user interface to the INIT, sending mode commands and displaying the data generated by the INIT.

The nature of the tool requires that the test case take a performance hit, typically about 140 instructions, plus two calls to the _MicroSeconds trap routine, for a total of about 220 instructions per trap, with selector based traps adding about 20 instructions. This is a fairly major hit and will bias the results of times for traps invoked from within other traps.

1.2 What MacsTime is Not

MacsTime is not a code coverage tool.

MacsTime is not a path analysis tool.

MacsTime is not a non-intrusive trap analysis tool. The tool does take an appreciable amount of the system to run. Figures of 30% have reached me, and I can easily see a larger hit for some pathological cases.

2.0 Macstime as a Self Contained Tool

The tool is available on the server "STM Central" (volume 'STM Tools') in the zone "ReadMe First!" in the folder "STM Tools:MacsTime". The current version is 2.0A2 Release 2. No other version should be used, as the previous versions are not feature complete.

Once you have a copy, drop the control panel into your system folder and restart the machine. The installation is done, and MacsTime is ready to produce timing data. Activating the control panel will cause the window displayed as Figure 1, below, to display, this provides a user interface to the INIT portion which loaded as part of the boot process.

Test execution is controlled by the set of buttons along the right side, and given parameters by the grid of text boxes in the right center under control of the radio buttons along the left side. The text boxes are initially unlabeled, as the startup mode runs under the control of the **Start** and **Stop** buttons. The boxes acquire labels according to the execution mode as the radio button mode changes. These modes are described below.

As a first example, clicking on the **Start** button with no other activity will begin a test of the Finder idle code. Clicking on the **Stop** button after a few seconds will result in a set of data being displayed. The data may be saved to disk by clicking on the **Save** button, which will cause a standard file dialog to appear. The **Clear** button will clear the data displayed on the screen and the internal data bases for the next run.

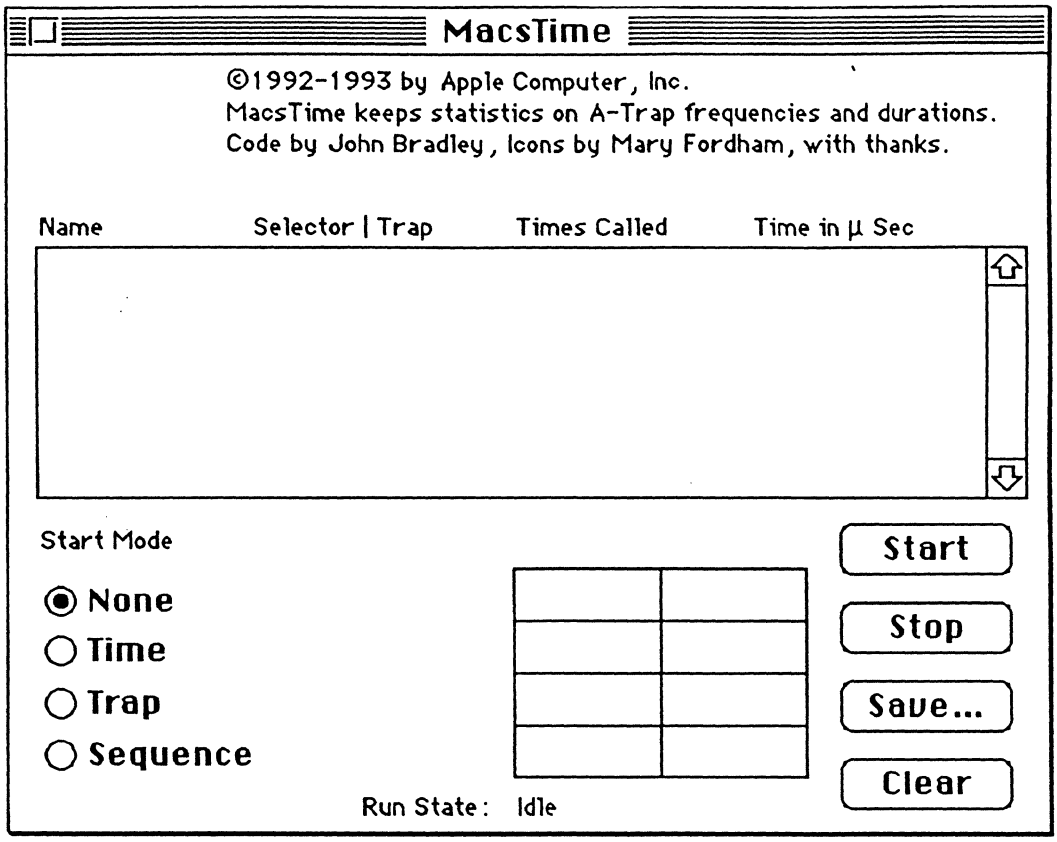


Figure 1

2.1 Controlling the Timing Mode

The radio buttons tell the INIT section of the program what start and stop criteria to use. As you change the radio selection, the labels on the text boxes will change to reflect the required parameters for the requested test. Note that the default case (above) has no labels on its text boxes, reflecting the fact that an unconditional start takes no parameters. There are NO default values to the radio button text fields, all values must be explicitly stated. The radio buttons work as follows:

None — Uses the **Start** and **Stop** buttons to initiate timing. Start will begin timing at the point of mouse release. Stop will cause timing to end on mouse release. The timing data will be displayed immediately on mouse release, as well. Since the text boxes are not used in this mode, they are not labeled, as in Figure 1, above.

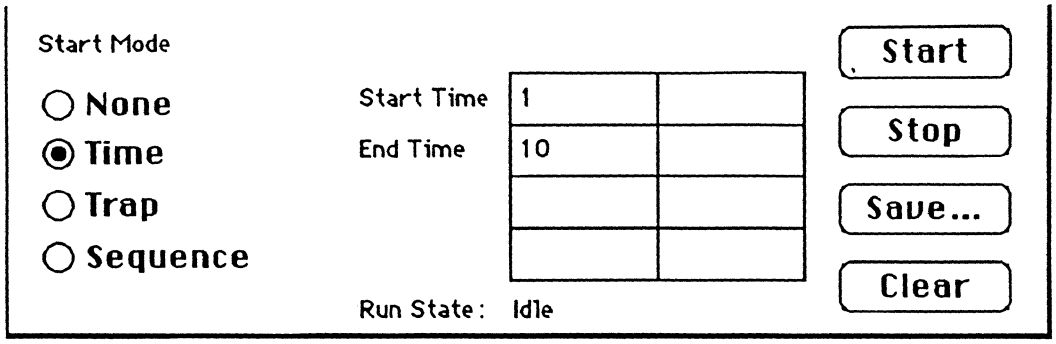


Figure 2

Time

Uses the two top left text boxes to receive two decimal integers. The upper is used as a delta from the time the Start button is selected to the actual start of trap timing. The lower is considered a test duration. Both are measured in seconds, and at the end of the test, the data is displayed automatically. Figure 2, above, illustrates a ten second run, starting one second from the time the user clicks on the start button.

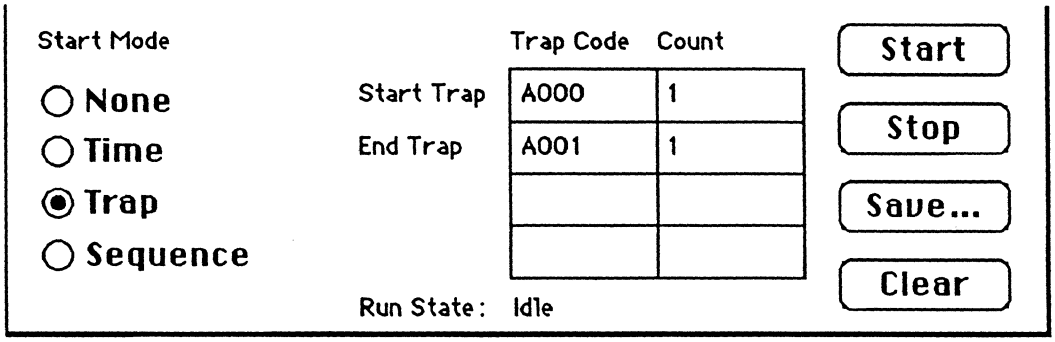


Figure 3

Trap

Receives data entered in the upper half of the grid to get the A-Trap value and count data. The left pair should contain the start and stop traps and the right pair contains the count data (which will typically be one). The INIT will begin timing when the start trap is seen for the 'count-th' time and end when the stop trap has been seen for its 'count-th' time. As usual, the data will be displayed automatically when the test ends. Figure 3 illustrates the setup for a run which will start on the first open call and end on the first close call performed after the start button is pressed.

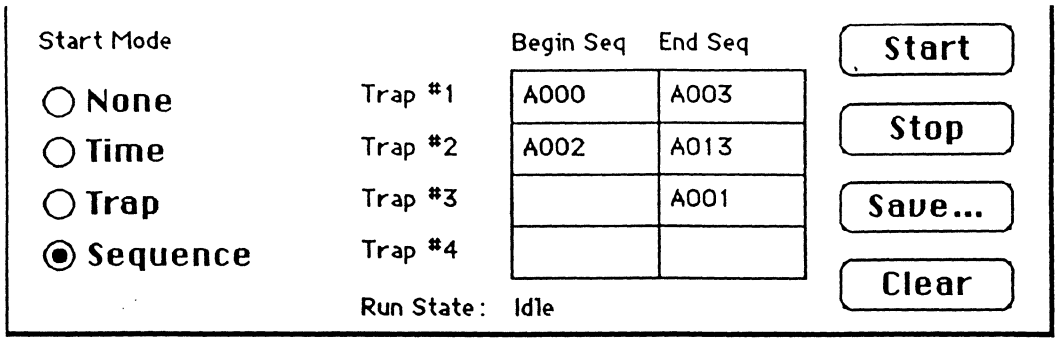


Figure 4

Sequence — This mode uses the entire grid to take two 1-4 trap sequences, one to begin the run (in the left column) and one to end it (in the right column). The sequences should be entered from the top of the column to the bottom with the first trap in the sequence at the top. The two sequences do not have to be the same length, but sequences of a single trap should use the Trap button, as it is a lower overhead method. A sequence must have no breaks (no blank entries) and will be searched for in order of the list, from top to bottom. Figure 5 gives a sequence pair, the start sequence is an Open, Read pair, the stop sequence is a Write, Flush, Close sequence. This will probably never occur in practice due to the traps invoked internally by these calls, so use this mode with care.

2.2 Executing a Test

Now you actually do the run. A timing run is started by clicking on the **Start** button. If the **None** button is set, then the INIT will start timing immediately. In any other mode the INIT will enter *Looking* mode. This means that the INIT is waiting for the startup conditions to be met. Once the conditions are met, the INIT will make the transition to *Working* mode, and from *Working* to *Done* when the end conditions are met.

2.2.1 Saving the Acquired Data

The **Save** button performs this function. It causes a standard file dialog to be shown and requests the output file name and folder. The output file is a tab delimited text file, designed to be input to any spreadsheet (both Excel and Wingz work for this). The first line is a set of column headers, the second and successive lines are trap statistics, one trap per line. Note: the default folder for the save file is the system folder.

2.2.2 Terminating a Test

Yes, it will stop a run in any state. Data may or may not be consistent at this point, but it generally will be if the current state was *Working* and will be displayed if so. Owing to a bug in the control panel, the wall clock time for the run may be displayed in the data area after a

run is ended by using the **Stop** button. This will occur only if no data had been collected when the button was clicked.

2.2.3 Clearing the Data Base

The **Clear** button clears the internal database, the parameter text strings and some internal state information. To prevent data loss, it can only be used during *Idle* or *Done* modes. Use the **Clear** button between runs to clear the database of count and time data so that the next runs data will not be added to the last. On the other hand, if several runs are to be summed, don't clear the database between them.

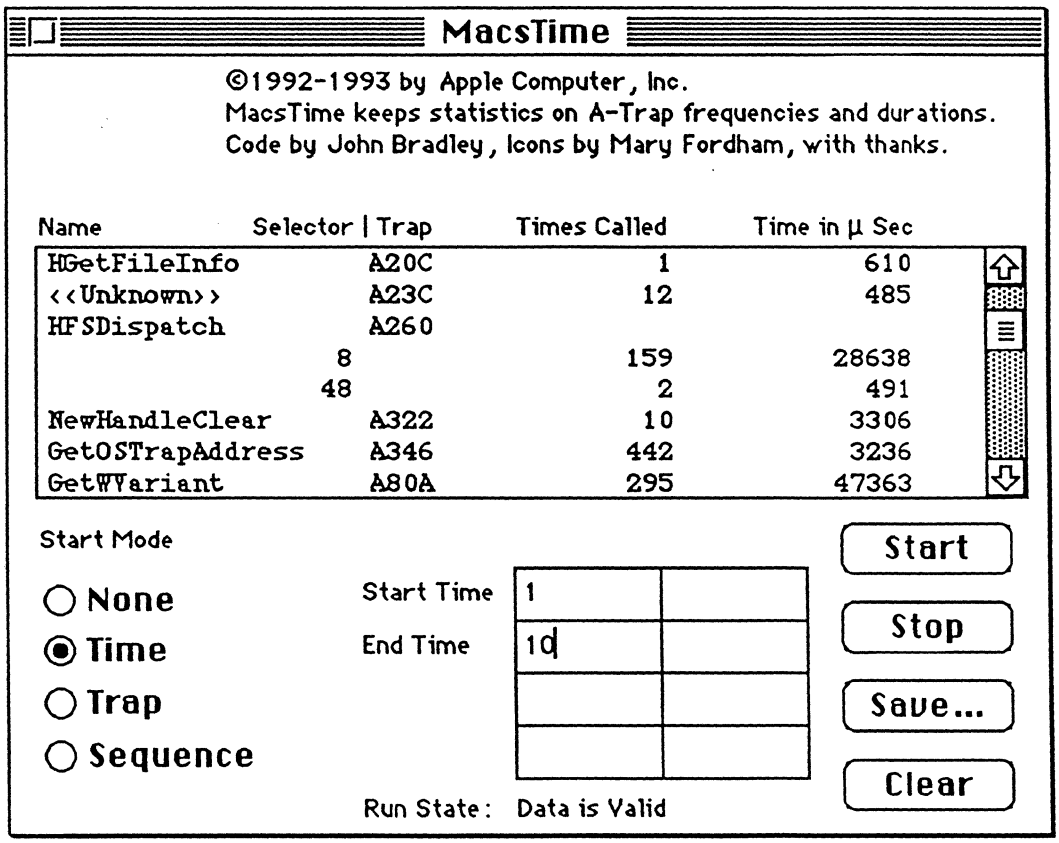


Figure 5

2.2.4 The "Run State" Indicator

The "Run State:" reflects the state of the INIT. On system boot, the INIT is in Idle state, the various other states are: Looking, Working and Done. The explanation for the modes is:

Idle	The INIT is simply passing all traps through to the A-Trap handler, recording no data and introducing about 20 instructions per trap.
Looking	The INIT has been given a 'Conditional Proceed' command. This causes the INIT to begin looking for the conditions to make a state transition to the Working state. Once found, the engine will transit to...

Working	The INIT is gathering data. It will continue to do so until either the completion criteria are found or the **Stop** button is clicked. In either case, the transition will be made to the Done state and the run's data will be displayed.
Data is Valid	The INIT is idling in this state, and data is guaranteed to be consistent. In this state, the database is displayed by the control panel if it is active.

2.3 Reading the Data Window

The data area (the large rectangle in the upper, center section of the window) is labeled, by column. Each of the columns displayed is part of the picture for a given trap.

The leftmost column contains the trap name for each trap. These names are taken from the file Traps.a. The trap name "<<Unknown>>" is used where the trap name was unavailable. Please send the names of any of these that you know to John Bradley, at x4-4677. The name for a selector-based trap is the name of the base trap and will probably be xxxDispatch, as done for HFSDispatch, above. The actual names for selector based traps are not currently in the database, but could be included if you have a need for it, so request selector naming if it would be useful to you.

The "Selector | Trap" column tag shows either the trap code as a four digit hex number (on the right) or the selector code for the last trap shown with a trap code (on the left). Thus, in Figure 1, the lines containing 8 and 48 in the selector field represent the PBGetFCBInfo and PBGetVolParms, respectively. These two traps are invoked via the A260 trap, and are selector based within that trap. The selector based traps are displayed in the manner so that selector s will be set off by having a blank name field and thus will be easily recognized. Each selector based trap has a line for each selector found during the run.

The "Times Called" displays a count of the times that any given trap was invoked. Only traps executed at machine level 0are counted, those executed from interrupt routines are not. The "Time in μ Sec" entry shows the sum of the times in micro-seconds for the trap, exclusive of the time spent in other traps. The inclusive time is available in the save file Time2 field.

2.4 Reading the Save File

The output file is a tab delimited text file set to be opened by TeachText. The file may be dropped onto Excel or Wingz to give an easily used spread-sheet of the data. A short sample of a run's data appears (with appropriate tab stops) as:

Name	Value	Selector	Count	Time1	Time2	Minimum	Maximum	Sum of Squares
Read	A002	0	106	300747	305618	1471	18922	2077642027
GetVol	A014	0	3	496	511	114	264	96616
SetVol	A015	0	6	860	882	126	164	124782
SetZone	A01B	0	3	36	36	11	13	434
DisposeHandle	A023	0	16	1709	1765	83	116	183363
SetHandleSize	A024	0	3	145	145	44	55	7077

The names and their meanings are as follows:

Name	The trap name, or the base name of a selector based trap. Since all non-selector based traps have a zero value in the Selector field, it is not possible to tell whether a trap is selector based by this file format unless the selector is non-zero. Unlike the display form, the name field is always stated for traps in the save file. This is to allow the file to be sorted without losing the trap name information.
Value	This is the value of the A-Trap in hex as executed. This is included so that traps which have several forms (such as NewPtr) or traps with selectors can be told apart after a sort operation.
Selector	The selector for a selector based trap, zero for all others. Most of the time, this is a place holder. The field is displayed in decimal. Note: traps with valid selectors of zero do exist, so be wary of dismissing zero selectors out of hand.
Count	The number of times this trap was executed during the course of the test. The various data values are stored as 32-bit unsigned values, and displayed as signed by Pascal (which does not have an unsigned type). If any of the counts or other values goes negative, then an overflow has occurred. The data for that value should not be used, since there is no way of knowing how many times the overflow occurred. Typically only the 'Sum of Squares' field will overflow.
Time1	The amount of time (measured in micro-seconds) that the trap spent in execution, exclusive of the time the trap spent in other traps. The time is obtained from the _Microseconds trap, which is called via JSR twice for each trap executed. Time spent in nested traps is added to the start time so that only time spent executing this trap is counted.
Time2	Similar to Time1, except that this time includes the time spent in other traps. Thus, it will always be greater than or equal to the Time1 value.
Minimum	The minimum value of the Time1 for all instances of a trap. Comparing this and the maximum value will give a good indication of that a bad worst case time exists for a trap, and thus it may be a candidate for optimization.
Maximum	Similar to the Minimum value, this is the largest Time1 value.
Sum of Squares	This is the sum of squares of the Time1 values. It is kept as a 32-bit unsigned value, but Pascal displays it as signed. However, since a sign change would indicate an overflow, negative values should not be corrected as there is no way to determine how many times the sign changed. This problem casts a doubt on the worth of the computation,

but being able to compute the standard deviation of the time is of value, even with this problem.

3.0 Programming for MacsTime Users

There are two ways to program the MacsTime INIT. The first, and simplest is to use a library of glue routines to perform the _Gestalt based calls necessary to drive the INIT. This library is supplied in source form with the package. The second method is to make these calls directly. Either way, the net result is that you are making calls into the INIT's control procedure and taking direct control away from the control panel. This relegates the control panel to being a convenient means of saving the results of a run. There is no requirement that the control panel even exist in memory during or after a run for the INIT to be used.

The INIT is commanded by a procedure linked in with it. This means that the procedure is loaded into the system heap. To obtain the address of the procedure, invoke _Gestalt with the selector 'maxt'. This will return, as the `long` result parameter, a procedure pointer. The procedure referred to by this pointer is of the form:

```
pascal short   MacsTimeControl(  short            command,
                                 unsigned long    P1,
                                 unsigned long    P2,
                                 unsigned long    P3,
                                 unsigned long    P4 );
```

The procedure is invoked using a statement of the form:

```
err = (*macsPtr)( <<Command>>, <<Param>>, <<Param>>, <<Param>>, <<Param>> );
```

This procedure issues the commands detailed below to the INIT, which checks them for errors and executes as commanded. Any parameter which is not used is ignored, but may **not** be omitted. The command effects are immediate, although the *Conditional* class of commands may not show any immediate change in behavior. The short returned by these commands is an OSErr in all cases except `cmdRunState` which uses the function result to return the current run mode of the INIT.

The command set for the INIT is:

cmdStartNow = 0	Returns either noErr or macsCmdErr (if a test is already running). No parameters used, the effect is immediate.
cmdStopNow = 4	Returns either noErr or macsCmdErr (if the tool is in Idle or similar state). No parameters used, the effect is immediate.
cmdCondStart = 8	Returns either noErr or macsCmdErr (if the tool is not in Idle state). No parameters used, the effect is immediate.

cmdRunState = 12

No parameters used, the run state result is returned as the function's return value. This is the only exception to the rule that these functions return an OSErr as the function result. The return results are:

sInitIdle = 1	the INIT is in idle state,
sInitLooking = 2	the INIT is looking for a set of start criteria to be fulfilled,
sInitRunning = 3	the INIT is running, taking trap times and counts and
sInitComplete = 4	the INIT is idling and the database is valid.

cmdSetCountTraps = 16

Returns either noErr or macsCmdErr (if a test is already running). The parameters are: P1 is the start trap code, P2 is the start count (timing will begin after the count-th trap is executed), P3 is the stop trap code and P4 is the stop trap count (counting will halt after the count-th trap is executed).

cmdSetStartSeq = 20

Returns either noErr or macsCmdErr (if a test is already running). Parameters P1 through P4 are the trap codes to look for to begin trap timing. Any codes unused should be passed in as zero. One to four traps may be specified, and the exact sequence must be issued in order to begin timing. *Note: this includes traps issued within other traps, so use this with care.*

cmdSetStopSeq = 24

Similar to the cmdSetStartSeq command, this specifies the trap sequence to stop timing on.

cmdSetStartStopTime = 28

Returns either noErr or macsCmdErr (if a test is already running). The P1 parameter specifies the number of ticks to wait before starting timing. The P2 parameter specifies the number of ticks the test will run. Both of these are specified in system timer ticks (about 1/60 th of a second). The engine uses the Ticks low memory global to determine when the timer has expired.

cmdSetRunPID = 32

Sets the process id for the test. The value is passed as the P1 parameter. The only traps timed will be those executed during the running of this process. This mode is independent of the other timing modes and is used to monitor a single process (such as the Finder). Returns either noErr or macsCmdErr (if a test is already running), or macsNoProcess (if the process does not currently exist).

cmdSetSingleTrap = 36	Sets the single trap to be timed. The value is passed as the P1 parameter. This is also independent of the other run modes and is used to time a single trap. In combination with cmdSetRunPID, this can be used to write a tool to test a single trap in isolation. Returns either noErr or macsCmdErr (if a test is currently running).
cmdClearCounts = 40	Clears the count data from a previous run. Used to clear the count data between run when a test case is composed of multiple runs. Returns either noErr or macsCmdErr (if a test is currently running).
cmdReturnDataPtrs = 44	Used at any time, this procedure returns a pair of pointers to the OSTrap and ToolBoxTrap data structures and the sizes of each in units of structure entries. The P1 and P2 parameters should be pointers to TrapPtr. These are used to store the addresses of the OSTrap and ToolBoxTrap tables, respectively. P3 and P4 should be pointers to short, used to store the OSTrap and ToolBoxTrap table sizes. The data structures are described in the MacsTime design document. The sole returned value is noErr.
cmdReturnClockTime = 48	This procedure returns the wall clock time during which the test case was run. All of the parameters are pointers to the various time values, as follows: P1 should be a pointer to a MicroTime structure which will contain the start time in μ-seconds of day of the start of timing, P2 is a pointer to a MicroTime structure which will contain the μ-seconds of day of the end of timing, P3 is a pointer to an unsigned long which will contain the time of start in clock ticks, P4 is a pointer to an unsigned long which will contain the time of test end in clock ticks. The errors which may be returned by this call are: noErr, or macsCmdErr (if a test is currently running).

The C Language templates, below, are prototypes for the glue routines used by the control panel. These detail the easy way to control INIT. Each procedure invokes _Gestalt to get the call address and then invokes the command function, passing the parameters as required by the caller. Full explanations of the functions are given with the command selectors, above. Note that the parameter set for each procedure has been reduced to that which is necessary for the command in question. The glue routines will provide any missing place holderparameters for you. The glue routines are supplied in source form with the software distribution.

```
pascal OSErr  CmdStartNow( );
pascal OSErr  CmdStopNow( );
pascal OSErr  CmdCondStart( );
pascal OSErr  CmdClearCounts( );
pascal short  CmdRunState( );
pascal OSErr  CmdSetCountTraps(       unsigned long     startTrap,
                                      unsigned long     startCount,
                                      unsigned long     stopTrap,
                                      unsigned long     stopCount);
pascal OSErr  CmdSetStartSeq(         unsigned long     Trap1,
                                      unsigned long     Trap2,
                                      unsigned long     Trap3,
                                      unsigned long     Trap4);
pascal OSErr  CmdSetStopSeq(          unsigned long     Trap1,
                                      unsigned long     Trap2,
                                      unsigned long     Trap3,
                                      unsigned long     Trap4);
pascal OSErr  CmdSetStartStopTime(    unsigned long     startTicks,
                                      unsigned long     stopTicks );
pascal OSErr  CmdSetRunPID(           unsigned long     PIDFirstLong,
                                      unsigned long     PIDSecondLong );
pascal OSErr  CmdSetSingleTrap(       unsigned long     onlyTrap );
pascal OSErr  CmdReturnDataPtrs(      TrapPtr           osTrap,
                                      TrapPtr           toolBoxTrap,
                                      unsigned long     *OSCount,
                                      unsigned long     *ToolBoxCount);
pascal OSErr  CmdReturnClockTime(     MicroTime         *startMS,
                                      unsigned long     *startTicks,
                                      MicroTime         *stopMS,
                                      unsigned long     *stopTicks );
```

3.1 Basic Program Flow of Control

Fairly simple. There is a standard sequence of functions to run a test. Build your test program or tool using variations on the sequence below.

CmdStopNow( );	Use this only if you're feeling paranoid. The system starts up in idle state, so this should never be necessary unless a previous test crashes.
CmdClearCounts( );	Clear any data currently left over from the last test case run.
CmdSetSingleTrap(0xA000);	Optional, used to monitor a single trap, in this case _Open. Use this to determine whether or not some trap is performing at speed. The Time2 field is useful in this case.

CmdSetRunPID(high, low);	Optional, used to restrict the timing to a single process.
CmdSetCountTraps(0xA000, 1, 0xA001, 1);	Optional, used to set the starting criteria. This is the hard part. Only one of the set (`CmdSetCountTraps`, `CmdSetStartStopTime`, `CmdSetStartSeq`) can be used on any single test. These require that the test be started using the `CmdCondStart` routine.
CmdCondStart( );	Start the search for stated begin condition. One of the set (`CmdSetCountTraps`, `CmdSetStartStopTime`, `CmdSetStartSeq`) must be issued before this command has any meaning. If you are writing an application that times a specific set of traps, then use the unconditional begin/end setup.
CmdStartNow( );	Start timing now. Use this to time specific scenarios under which you have complete control. The corresponding call `CmdStopNow` will halt the test when it is complete.
while (`CmdRunState()` != sInitComplete) DoEventStuff();	Use a sequence like this to wait for the completion of a test that uses the conditional calls. Or simple generate the calls you need to time internal to the test code.
CmdStopNow( );	This will halt the run unconditionally. As above, use this to halt timing in a scenario under which you have complete control.
CmdReturnDataPtrs(&OS,&Tool,&C1,&C2);	Use this to return a set of pointers to the internal database. The pointers are only valid when counting is not being done, so bear this in mind when writing applications which use the database directly.
CmdReturnClockTime(&T1,&T2,&T3,&T4);	Use this to return the wall clock time for the start and end of a run. This is recorded from the time of timing start to the time of timing end. It is returned in both micro-seconds (T1 and T3) and clock ticks (T2 and T4).

3.2 Internal Data Structures

The internal structures are fairly simple, and designed for speed. The first structure is used to hold a μ-second time of day. This is the value returned from the Time Manager routine _MicroSeconds, and thus is in true μ-seconds (as opposed to the 1.2+ micro-second units which the VIA chip returns). It is returned to the caller by the CmdReturnClockTime() call, otherwise the programmer will never see values of this type.

```
typedef struct  microTime
{
        unsigned long   highTime;
        unsigned long   lowTime;
}       MicroTime;
```

The structure, below, contains the statistical data for a trap. The 'time' value contains the sum of times spent in this trap, exclusive of the time spent in other traps. 'fullTotal' contains the time spent in the trap, including the traps which this one called. 'timeSquared' contains the sum of squares for the 'time' value, use this to compute the standard deviation of the time spent in the trap. 'count' is the number of times that the trap was executed. 'maximum' and 'minimum' hold the largest and smallest trap time in 'time' terms.

```
typedef struct data
{
        unsigned long           time;
        unsigned long           fullTotal;
        unsigned long           timeSquared;
        unsigned long           count;
        unsigned long           minimum;
        unsigned long           maximum;
}       TimingData;
```

The Selector blocks are initially held in a free list, linked through the next pointer. As they are needed, the entries are removed from the list and linked onto the selector chain for the A-Trap which is used as the basis for the selector. When a selector trap is executed, the list associated with the trap is searched. If the selector is already in the list, then a pointer to the selector block is returned. If not, then the following sequence of operations occurs: a selector block is removed from the free list, the block is added to the selector list associated with the trap and a pointer to the selector block is returned to the main part of the tool.

```
typedef struct  selectorBlock
{
        TimingData              data;
        struct selectorBlock    *next;
        struct selectorBlock    *prev;
        unsigned long           selector;
}       SelectorBlock;
typedef SelectorBlock           * SelectPtr;
typedef SelectPtr               * SelectHandle;
```

This is the main data structure for the program. Two blocks of these structures are allocated as part of system startup, one each for OS Traps and Toolbox Traps. The data portion of this record contains the information generated for each trap executed as the test case runs. The head, tail and listCount parameters are used to track the selector driven calls. If the listCount field is non zero, then the selector list is non-empty and should be searched for data when the data is being read out. Note: the CmdClearCounts does not free Selector entries, it just clears the data in them. This means that if the listCount field is non-zero, there may be no data, because all of the selector blocks could be blank.

```
typedef struct trapEntry
{
    TimingData          data;
    SelectorBlock       * head;
    SelectorBlock       * tail;
    short               listCount;
}   TrapEntry;
typedef TrapEntry       * TrapPtr;
typedef TrapPtr         * TrapHandle;
```

3.3 Accessing the Data Structures

The CmdReturnDataPtrs() routine returns two pointers of type TrapPtr and the entry counts for each. Use these pointers and counts to traverse the arrays in a manner similar to:

```
for (i=0; i<C1; ++i)
{
    if ( (P1[i].data.count == 0) && (P1[i].listCount == 0)
        continue;                   // no data here

    if (P1[i].data.count != 0)
    {
        ... Do something, non-selector trap found ...

        continue;
    };
    if ( IsDataPresent(&P1[i]) )                // we may have selector data
    {
        …Do something, selector data present …

        continue;
    };
};
```

IsDataPresent() is a dummy Boolean routine (not provided) which searches for non-empty entries in the selector list. This structure will allow you to search a trap list fairly quickly and extract the data you need from it.

Application Performance and the ToolBox

Jim Gochee, PowerPC
1/30/93
Rev 3

Overview

The true performance of any machine is measured by how fast its applications run, not necessarily how fast the CPU is. PowerPC is no exception. While the 601 is 3 to 4 times faster than a 25mhz 68040 (integer specmarks), applications that are emulated, or applications that call an emulated ToolBox, may see only a fraction of this horsepower. On average, native code executes 10 times faster than emulated code, which means an emulated application calling an emulated ToolBox would be 10 times slower than a fully native machine. Because the first PowerPC release will have significant amounts of emulated code, it is important to understand how fast applications can be expected to run in a mixed code environment. By varying the ratio of emulated instructions to native instructions, we can get a feeling for the machine's overall performance.

Applications that execute 30% emulated code and 70% native code will be 3.7 times slower than a fully native machine, or at the speed of a 25mhz 68040. However, as the amount of emulated code approaches 0%, Amdahl's law points out that gains will be offset by the high cost of executing emulated code. For instance, even if an application executes 90% native code, it will only be 2 times as fast as a 25mhz 68040.

The Facts

This table shows the effect of the equation:

NATIVE + (10)EMULATED = AVG_NATIVE_INSTRUCTIONS

NATIVE is the percentage of time spent executing native code, EMULATED is the percentage of time spent executing emulated code, and AVG_NATIVE_INSTRUCTIONS is the average number of native instructions per mixed instruction type. EMULATED is multiplied by 10 because each emulated instruction takes around 10 cycles, whereas each native instruction takes one cycle. At 30% emulated code, the machine should perform as well as a 25mhz 68040.

% Emulated Code	Avg Native Instructions
0	1
1	1.09
2	1.18
3	1.27
4	1.36
5	1.45
10	1.9
15	2.35
20	2.8
25	3.25
30	3.7
35	4.15
40	4.6
45	5.05
50	5.5
55	5.95
60	6.4

65	6.85
70	7.3
75	7.75
80	8.2
85	8.65
90	9.1
95	9.55
100	10

Emulated Application Performance

68k applications will probably not reach 25mhz 68040 performance by V0, though they will come close. Studies show that applications generally spend 60-80% of their time in the ToolBox. An average application will execute its own 68k code (30%) plus some amount of 68k ToolBox code. Given two scenarios, one where 70% of the executed ToolBox instructions are native, and another where 90% are native, the average emulated application will execute at 74% of a 25mhz 68040.

$$(10 * .30) + ((1 * .49) + (10 * .21)) = 5.59$$

$$(10 * .30) + ((1 * .63) + (10 * .07)) = 4.33$$

Average = 4.96, compared with 3.7 for a 25mhz 68040, (3.7/4.96) = 74%

Native Application Performance

For native applications to fully realize the potential of the PowerPC, it is imperative that there is as little 68k code as possible. This means that almost all ToolBox routines (and the routines they in turn call) should be native. Since this is unrealistic for V0, there will be a portion of the ToolBox that is native and the rest will be 68k. To see how quickly the performance of a native application can deteriorate as the percentage of 68k code increases, let's assume that applications execute 70% of their instructions in the ToolBox. The worst case scenario is that the entire ToolBox is emulated, which from the equation below shows the machine running 7.3 times slower than it would if no 68k code were executed. Another way to look at it is that the native application would be 2 times slower than a 25mhz 68040. This is the worst case.

$$(1 * .30) + (10 * .70) = 7.3$$

Now suppose that enough of the ToolBox is ported so that 80% of the instructions executed in the ToolBox are native. We still find that the machine is more than twice as slow as its potential and only 61% faster than a 25mhz 68040.

$$(1 * .30) + ((1 * .56) + (10 * .14)) = 2.26$$

Just in case you think Amdahl's law is horse radish, assume that 95% of the instructions executed in the ToolBox are native. We still see that the machine could go approximately 1/3 faster if all the ToolBox was native.

$$(1 * .30) + ((1 * .665) + (10 * .035)) = 1.315$$

Context Switches

In the previous calculations, it was assumed that some fraction of the instructions were 68k and some 601. What was left out was the overhead of the machine switching back and forth between the two code types. This switch incurs a penalty of 10-14μ, or 500-750 native instructions! Not only is it important to keep the amount of emulated code to a minimum, but it is crucial to limit the number of switches. For this reason, ToolBox call chains must be completely ported.

The Plan

Based on the assumption that we have a limited time to port ToolBox routines, it is imperative that we focus our energy wisely. The first step is to fully understand how applications interact with the ToolBox. By examining performance sensitive sections of native applications with the PowerProfiler, we can pinpoint routines to port. Also, the PowerProfiler will enable us to track context switches so that we can be 100% guaranteed that our execution paths stay native. Hopefully, we will have areas of the ToolBox where 100% of the code is native, and other areas where 0% of the code will be native. Decisions will have to be made about which areas of performance are most critical to the user.

While it appears that 68k applications should perform well compared to a 25mhz 68040, native applications will be severely limited unless they access a native ToolBox. We will spend much of our energy analyzing the Inside Track applications in search of common patterns and behaviors. With luck and lots of porting we hope to accelerate speed sensitive portions of these applications and give a sense for what the future of RISC will look like.

Current Status

We are currently focused on porting the ToolBox in the areas of Quickdraw and text display. All high level APIs are to be ported along with complete call chains. Graphical output is usually the most speed sensitive part of the ToolBox, so we hope to get big wins from this. The PowerProfiler is being used on native applications created by MSD and MSSW to determine what to port next. Our next step is to get Inside Track applications as they are ported native. Guidance from Pierre Cesarini and Jordan Mattson will help keep us aware of weak spots in the system and areas where performance is critical. Jordan will coordinate with developers to get feedback as concerning where they think their applications should perform well.

60x System Software Interface Differences

Rich Witek
April 23, 1993
V1.3

Introduction

The bulk of the software interface to the 604 processors is common across the 601, 603, and 604 chips. The differences are grouped into three sets, each covered in a following section. The first set are the differences visible to both privileged and nonprivileged modes of the processor. The second set are differences visible only to privileged mode software. The third set of differences are where either the 601 was too far along when the change was made or the 601 included some power support to aid in moving the AIX users from Power to Power PC. The 601 differences were done at request of Apple or with Apples agreement.

There is a lot of concern over the variations between the chips in the 60x family. Much of this concern is based on the history of the changes between chips in the 68K family. The situation with the PowerPC chips is different in that there is an architecture that all chips are following.

Non Privilege Mode Software

Cache organization

The 601 has a combined Instruction and data cache while the 603 and 604 have split instruction and data caches. The PowerPC architecture allows the instruction cache to be incoherent with the data cache. The PowerPC architecture encourages operating systems to provide a system service to make the Istream and Dstream coherent, see PowerPC Virtual Environment Architecture Book II, Section 3.2.1. The 601 by having a combined instruction and data cache allows one to "CHEAT" and not call the system service and have code that will work on a 601 and not work on a 603/604.

Cache size

The 601 has a 32 Kbyte combined instruction and data cache. The 603 has an 8 Kbyte instruction and an 8 Kbyte data cache. The 604 has an 16 Kbyte instruction and an 16 Kbyte data cache. An application that is tuned to the 32 KByte cache of the 601 will see very different performance from the smaller 603 cache. The 68K emulator is a good example of that. We need to set expectations correctly for the 603 or else people will be surprised in a bad way.

Cache control instructions

The 603 does not broadcast the cache ops icba, dcbt, dcbtst, dcbz, dcbst, and dcbf, on the 60x bus. Since the 603 is targeted at single processor systems this should not be an issue. This will be a problem if

the 603 is used in an multi processor system and the software depends on the cache ops working across multiple CPUs.

Unaligned accesses

The 603 does alignment checks on a word (32 bit) basis. This means that access to unaligned words that do not cross a double word boundary will cause two bus transactions on the 603 and one on the 604.

Word gathering

The 604 does word gathering in load and store multiple even to a cache inhibited space. This means that load and store multiples should not be used to access device registers.

Scheduling

The 601, 603, and 604 need different basic block scheduling for best performance. Compilers should provide a common and specific schedulers for these machines.

Privileged Mode Software

Translation buffer control instructions

The 603 does not broadcast the cache ops tlbie, tlbia, tlbsync on the 60x bus. Since the 603 is targeted at single processor systems this should not be an issue. This will be a problem if the 603 is used in an multi processor system and the software depends on the tlb ops working across multiple CPUs.

Software Table Walk

The 603 does translation buffer reload with a software routine while the 601 and 604 have the hardware do reload. The reload routines use a vector not used on the 601 or 604 so the reload code can be included in the ROM of all three machines even though it is only used on the 603. The PowerPC architecture allows the special purpose registers SRR0 and SRR1 to be destroyed when a translation buffer miss occurs. The 603 takes advantage of this while the 601 and 604 do not. This means that if 601 system software "CHEATS" and does not always save/restore SRR0 and SRR1 on an interrupt that turns relocation on it will break on the 603.

Book 4 SPRs

The implementation specific SPRs have been defined to be common as much as is possible across the four chips. The following tables show uses of the machine specific SPRs. There are some differences in these registers so software should isolate access to them to routines that can contain per CPU type code for each function. In defining these registers

the trade off was to make the 603, 604, and 620 common if they needed to differ from 601.

Book 4 SPRs:

SPR	What	601	603	604	620
976	DMISS		√		
977	DCMP		√		
978	HASH1		√		
979	HASH2		√		
980	IMISS		√		
981	ICMP		√		
982	RPA		√		
1008	HID0	√	√	√	√
1009	HID1	√			
1010	IABR	√	√	√	√
1013	DABR	√	√	√	√
1016	BUSSCR				√
1017	L2CSR				√
1018	L2ECCR				√
1023	PIR	√		√	√

HID0 for 603, 604, and 620:

Bits	Name	603	604	620
0..7	Enables	√	√	√
0	EMCP - Machine Check Pin Enable	√		√
1	EM - Machine Check checkstop Enable	√	√	√
2	EBA -Bus address parity checking	√	√	√
3	EPD - Bus data parity checking enable	√	√	√
4	EPP - PIO/Direct Store error checking enable	√		
5	EICE - Enable ICE outputs	√		
6	DCLK - Disable external test clock	√		
7	PAR - Disable precharge of ARTRY /shared pins	√		
8..11	Power Management modes			
8	Doze	√		
9	Nap	√		
10	Sleep	√		
12..15	Error Logging			
16..23	Cache Control			
16	ICE - Instruction cache enable	√	√	√
17	DCE - Data cache enable	√	√	√

18	ILOCK - Instruction cache lock	√	√?	
19	DLOCK - data cache lock	√	√?	
20	ICI - Instruction cache invalidate	√	√	√
21	DCI - Data Cache invalidate	√	√	√
22	DFWT - Data Force Write Thru			√
23	RISEG - read I seg	√		
24..31	Run Modes			
24	Dispatch Mode (0 - single instruction, 1 - normal)		√	√
25..26	Branch Prediction modes			√
27..28	Instruction Fetch Modes			√
31	NOOPTI - Noop Touch Instructions)	√		

HID0 for 601:

Bit	Definition
0	e - master Checkstop Enable
1	s - microcode Selftest Checkstop Latch
2	m - checkstop following Mchine check with me=0
3	td - multi-side hit in tlb
4	cd - multi-side hit in the cache
5	sh - sequencer hang
6	dt - dispatch timeout
7	ba - bus address parity error
8	bb - bus data parity error
9	cp - cache parity error
10	iu - invalid microcode instruciton
11	pp - pio bus protocol error
12:14	reserved
15	es - enable for ucode selftest checkstop
16	em - enable for machine check checkstop
17	etd - enable for tlb checkstop
18	ecd - enable for cache directory checkstop
19	esh - enable for sequencer hang checkstop
20	edt - enable for dispatch timeout checkstop
21	eba - enable for bus address parity checkstop
22	ebd - enable for bus data parity checkstop
23	ecp - enable for cache parity checkstop
24	eiu - enable for invalid ucode instruction checkstop
25	epp - enable for pio bus protocol checkstop
26:29	reserved
30	emc - error detected in main cache during initialization

31	reserved

HID1:
only used in 601

IABR register

bits	What	601	603	604	620
0..29	Address	√	√	√	√
30	IE - IBR Enable		√	√	√
31	BT - IBR translation Enable			√	√

DABR:

bits	What	601	603	604	620
0..28	Address	√	√	√	√
29	BT Traslation Enable			√	√
30	DW Write Enable	√	√	√	√
31	DR Read Enable	√		√	√

PIR:

bits	What	601	603	604	620
27	Implemented				√
28..31	Implemented	√		√	√

601 variances

Block Address Translation

PowerPC now contains four instruction and four data Block Address Translation registers (BATs). This is different than when the 601 was being done. The 601 contains only four BATs. The 601 also contains a support for special direct store segments. These are segments where the T-bit is set and the BUID field is x07F. The memory management code will need to deal with both the 601 and PowerPC structures.

Time Base

PowerPC changed the definition of the time base from a nanosecond counter to a cycle counter. The instructions to access the time base were done in a way to allow emulation of the cycle counter on all systems. System software should provide a routine to access the timebase that can be different on different CPUs that returns cycle counts. The cycle count can be emulated on the 601 using the nanosecond counter.

Extra instructions

The 601 supports several Power instructions in addition to the PowerPC instructions. This was done to ease transition from Power to PowerPC. The PowerPC assemblers and compilers should not generate anything but PowerPC instructions.

Edit History

24-Dec-1992	First Pass
18-Jan-1993	Add load/store multiple gathering on 604
	Add 603 unaligned word access rules
	Fix typos
20-Jan-1993	Remove timebase 603/604 difference
23-Apr-1993	Fix few typos for software group

DTS PowerPC Porting Tips

Dave Radcliffe

Introduction

So, you want to port your application as a native PowerPC application. What are you in for? This document may help you answer that question. Rather than being a formal porting guide, this document is practical tips and advice based on my "hands-on" experience porting the DTS sample application, Kibitz. While some of this may be covered in the more official documentation, this document touches on practical issues, such as "what does that weird compiler error mean?"

This document assumes that most of your code is written in fairly portable (i.e. ANSI standard) C. It does not cover issues of porting massive amounts of Pascal or assembly. What it does cover is the build process, i.e., getting MPW to compiler your code using the IBM C compiler, issues using the IBM C compiler, and issues with the one major interface change – using RoutineDescriptors instead of ProcPtrs.

Build Process

The first step is to modify your build process for the RS/6000 (herein referred to simply as the "Unix" system). If you currently build using MPW, this will be pretty easy, but if you use Think C, you should very strongly consider porting it to MPW first. While both Unix and MPW have similar build facilities ("make") the build process is easist to control from MPW. Since Think now supports MPW headers and calling conventions, conversion from Think C to MPW C should not be too difficult and can be done such that a single set of sources can compile with both Think C, MPW C (and after this port, RS/6000 C). You can use the MPW CreateMake tool to assist creating the Makefile for building under MPW.

Pathnames

A short word is in order about pathnames. Unix is a lot pickier about file and pathnames than the Macintosh. You can save yourself a lot of trouble by simplifying things on the Macintosh side before moving them to Unix. Unix filenames are case sensitive and only 7-bit ASCII characters are allowed. In addition, non-alphanumeric characters often have special meanings, so stick to upper and lower case alphanumeric characters and don't use spaces or slashes ("/").

Here are some examples of ways this can mess you up.

With MPW build scripts, you can ask to build the Kibitz application by asking for either "Kibitz", or "kibitz", but with Unix, asking for a build of "kibitz" when the Makefile is called "Kibitz.make" doesn't work because the build script will look for "kibitz.make" and the name won't be found.

Similar problems may exist if files within the Makefile or #include files in the source are not consistently cased. As another example,

```
#include <Quickdraw.h>
```

works, but

```
#include <QuickDraw.h>
```

doesn't.

Another possible source of confusion is similarity of names between the standard Unix include files in /usr/include and the Macintosh interface files provided. For example there is the standard Unix include file /usr/include/time.h and the Macintosh include file Time.h. If the case is wrong, the incorrect include file might be found, leading to all sorts of strange and wonderful errors. So beware of the following include file names, which have lowercase cousins in the Unix environment:

Assert.h	Float.h	Memory.h	StdDef.h	Strings.h
CType.h	Limits.h	SetJmp.h	StdIO.h	Time.h
ErrNo.h	Locale.h	Signal.h	StdLib.h	Values.h
FCntl.h	Math.h	StdArg.h	String.h	

Another difference between Macintosh and Unix is pathname specification. On the Macintosh, path elements are delimited with ":", while on Unix they are delimited with "/". If you use the remote tool (supplied with the seed tools), it attempts to minimize this difference by doing pathname conversion for you.

Makefile

Besides the issue of filename case dependency mentioned above, there are other modifications necessary to allow your Makefile to control the build process on Unix. The first is to override the default use of the MPW C compiler. You can do this by adding a line:

```
C       =       remote cmac
```

This tells MPW to invoke the remote tool to remotely issue the cmac compile command on the Unix. cmac is just the IBM xlc compiler with Apple extensions, so refer to xlc documentation for more information on cmac.

If you have custom build rules in your Makefile, be sure it symbolically references the C compiler, i.e.:

```
.c.o  f      .c
      {C} {COptions} {DepDir}{Default}.c -o {TargDir}{Default}.c.o
```

{C} tells MPW to use the value of the C variable (remote cmac in this case) when compiling .c files.

The IBM C compiler has completely different compiler options than MPW C so you will need to override the COptions variable, for example:

```
COptions     =      -c -I :::NativeInterfaces:Public: -Dapplec
```

This example illustrates some of the commonly used IBM C compiler options. "-c" tells the compiler to stop after creating the object file. Without the option, the compiler attempts to link the file as well. "-I" specifies paths to interface file directories. It is similar to the "-i" MPW C option, but uppercase "I" must be used. Note that we can continue to use Macintosh style pathnames because the remote tool will do the translation for us. The "-D" option defines preprocessor constants as though we had used #define statements. So "-Dapplec" is equivalent to:

```
#define applec
```

[There is more to add here on linking and the use of library files, but I haven't actually tried that yet, so I don't know what's involved -- DR]

Building

Once you have modified the Makefile, you can begin building just as you would under MPW. One advantage of using MPW and the remote tool to control the build process is that remote filters error output from the compiler and generates MPW "File" commands that you can execute to open source files and locate errors.

If you make changes to source files using MPW, you **must remember to save changes before compiling.** Unlike MPW C, which is tightly coupled to the MPW Shell, cmac is completely separate and runs on a different machine. MPW is not in control of file I/O, so changes must be saved to the Unix system before they can be seen by the Unix compiler.

Compiler issues

pascal functions

The IBM compiler has been modified to accept the "pascal" keyword. But when the IBM compiler encounters this keyword, it does***absolutely nothing***. Unlike MPW C where the pascal keyword alters parameter ordering and changes how some parameters are passed, "pascal" with the IBM compiler is simply ignored. This has some subtle consequences. For example, consider the following AppleEvent handler:

```
pascal OSErr DoAEAnswer(AppleEvent message, AppleEvent reply, long refcon);
```

In Pascal, an AppleEvent is a record larger than 4 bytes, and so is automatically passed by reference. MPW C, since DoAEAnswer is declared pascal, will handle the parameter in the same way. But cmac will treat it as a standard C struct and pass it by value, so if DoAEAnswer were called by the AppleEvent manager, bad things would happen.

In this case you must explicitly make these parameters pointers, i.e.:

```
pascal OSErr DoAEAnswer(AppleEvent *message, AppleEvent *reply, long refcon);
```

The new interfaces now declare special ProcPtrs that specify the correct parameters, e.g.:

```
typedef pascal OSErr (*EventHandlerProcPtr)(const AppleEvent *theAppleEvent,
      const AppleEvent *reply, long handleRefcon);
```

Unfortunately, in most cases you will now be coercing those special ProcPtrs (such as `EventHandlerProcPtr`) into normal ProcPtrs for calls to NewRoutineDescriptor (see below), which means typechecking will be lost. So, double check all of your callback routines.

Assembly wrapper routines

The current Mac toolbox makes all sorts of callbacks into developer code. Many of these have weird calling conventions, such as parameters in registers. These go away when you are called by the PowerPC toolbox, which will always call you with standard C calling conventions. So some of your assembly code probably consists of simple assembly wrapper routines that push parameters around and call C routines. The good news is that now you can just pass the toolbox the C routine directly; see the interfaces for the new calling conventions. The bad news is this is

not backward compatible to the current 68K API, so you will have to special case these modifications in your code.

For example, the TextEdit caretHook routine gets called in a truly bizarre way, with a pointer to the Rect containing the caret on top of the stack (not the return address!) and with a pointer to a locked edit record in register A3. But with the PowerPC toolbox, you can now just declared a standard C routine:

```
pascal void MyCaretHook (Rect boundsRect, struct TERec *pTE);
```

There is also a corresponding typedef, which is useful to help figure out the routine parameters:

```
typedef pascal void (*CaretHookProcPtr)(Rect boundsRect, struct TERec *pTE);
```

Compiler ideosyncracies

Every compiler has its foibles. Here are a few I ran into:

The construct

```
#ifdef foo
  ...some stuff
#endif foo
```

is not strict ANSI C. ANSI C specifies nothing other than a newline after #endif. MPW C just throws the rest of the line away. cmac, on the other hand, complains. cmac does allow comments, which also seems like a violation of the standard. For example:

```
#ifdef foo
  ...some stuff
#endif // foo
```

works.

Another warning issued by the compiler is sure to drive you nuts. Use of OSType values, such as `'TEXT'` causes the following warning:

```
1506-076 (W) Character constant has more than one character.  Rightmost four
characters are used.
```

This can generally be ignored as the compiler does the right thing. One case where it can't be ignored is the sequence '????' (and similar sequences starting with '??'). The compiler confuses this with ANSI C trigraph sequences and generates the following error message (in addition to the one above):

```
506-209 (S) Character constants must be ended before the end of a line.
```

Use `'\?\?\?\?'` or `0x3f3f3f3f` instead.

You can turn off warnings with the -w compiler option, but it is probably best not to do this until you are sure all such warnings are harmless.

Miscellaneous tips

The compiler often generates multiple errors messages for a single error. The first error is often a generic "syntax error" while the subsequent error messages are more meaningful. For example:

```
"Utilities.h", line 140.15: 1506-046 (S) Syntax error.
"Utilities.h", line 140.15: 1506-081 (S) Discarding previously defined typedef
identifier: RectPtr
```

So, it pays to pay attention to line numbers in the error messages and look for multiple messages.

RoutineDescriptors

The single biggest change developers will have to make to their code is converting ProcPtrs to RoutineDescriptors. Every place in the interfaces where a type of ProcPtr was declared, Apple added a similar declaration of type RoutineDescriptor.

Definition RoutineDescriptor: a ProcPtr useful on more than one chip architecture.

The Macintosh Toolbox relies heavily on ProcPtrs. The toolbox on PowerPC will consist of a mixture of both 68K and PowerPC code. We needed a generic way of calling code that could be either PowerPC or 68K, from code that could be either PowerPC or 68K. RoutineDescriptors describe not only the address of a routine, but its parameters and calling conventions as well. That's enough information to get us from one type of code to the other and back.

RoutineDescriptors will work with the 68K API. We encourage you to change your code to support RoutineDescriptors, it will work in 68K-land as well.

A RoutineDescriptor is a pointer to a private data structure that, in addition to containing the function reference, carries additional information on parameter passing and return values. The internal workings of this data structure are not important (indeed, the data structure is undocumented to allow us to change it in the future), so it should simply be thought of as a function reference. This makes a RoutineDescriptor a generic version of a ProcPtr, but it's not quite as easy to use, because you must manage allocating and deallocating of RoutineDescriptors, while a ProcPtr is just an address you can pass around (actually, your compiler has always had the responsibility for this, but we're not changing the compilers to support RoutineDescriptors specifically).

A lot of thought needs to go into converting ProcPtrs into RoutineDescriptors. With some care it is possible to get code that works right when compiled for either PowerPC or 68K. But if you're not careful, subtle bugs can creep in.

Recommended usage

The simplest thing is to allocate a global RoutineDescriptor and if it's never been initialized, initialize it with NewRoutineDescriptor and forget about it:

in your initialization code:

```
if (!gVActionDesc)
      gVActionDesc = NewRoutineDescriptor ((ProcPtr)VActionProc,
            rdControlActionProcInfo, kCodeTypeCurrentWorld);
```

when you use your RoutineDescriptor:

```
TrackControl(ctlHit, mouseLoc, gVActionDesc);
```

In the 68K world, a routine descriptor is basically a ProcPtr, so gVActionDesc just gets initialized with VActionProc. Besides the problem of whether you have global variables (in other words, can you even have gVActionDesc?), there is the problem of relocation of VActionProc. Creating a static reference to VActionProc works fine as long as VActionProc is referenced via A5, or is in a permanently resident segment, but it can fail if VActionProc is an intra-segment reference and the segment gets unloaded and reloaded. So beware of code compiled with the -b or -b2 MPW C options. When in doubt, you can use SetTheProc() to update the RoutineDescriptor with the correct function address.

Scope

Another problem with RoutineDescriptors is scope. Be careful about leaving dangling RoutineDescriptors. If you create RoutineDescriptors on the fly, dispose of them when you are done. You should only need to do this for code which you load and execute. You should not do this for your normal program code which you can make resident for the duration of the application.

For example, consider a procedure doing asynchronous I/O and using a completion routine. If a local RoutineDescriptor variable is declared to replace the completion routine ProcPtr, the variable reference could be lost when the procedure exits although the I/O itself has not completed. So, beware of creating new RoutineDescriptors and not disposing them and beware of disposing them before the system is done with them. That's why it is best to make RoutineDescriptors global or static so you don't have to worry about it.

NOTE -- Not all ProcPtrs need to be converted to RoutineDescriptors. If you know the function is going to be called by code of the same type (e.g. PowerPC code calling a PowerPC function), then a ProcPtr works just fine. But if there's any chance at all of the ProcPtr being passed to a toolbox routine, make it a RoutineDescriptor.

☐ OK as is ☐ OK with Changes ☐ Not OK (You *MUST* Justify this)

10+ Commandments
Overview

Written by: Rich Collyer & Dave Radcliffe June 1993

Have you ever wondered what all of the rules are to programming the Macintosh? Have you ever wondered if you application is going to have problems with the future Macintosh OSs and hardware? Below are the most likely gotchas which are likely to getcha in the not too distant future. Some will affect your application performance, others will affect whether your application will run at all. Take a close look and if you see something which might getcha, then it is time for you to start fixing that code; but Don't Panic!! You're not going to get out of bed one morning and find that the issues we discuss here have bitten you in the night. However, these issues seriously affect the ability of Apple to transform the Macintosh operating system into a modern O/S. As you write new code, or review old code, be aware of these issues. If you can't deal with them immediately, at least flag problem areas with appropriate comments so you can fix them in the future. By doing so, you'll help Apple bring you those modern O/S features you've been screaming for that much sooner.

Topics

- Rules to Compatible Macintosh Programming
- To Live or Die in the Macintosh OS
- How to make your Application run into the late Nineties

---

1) Write in ANSI C or C++

This is a bit of a religious issue. There are some people who I have heard say that they will be dead and cold before their assembler can be taken from their fist. Unfortunately, assembly code is very hard to port to new CPUs and Pascal is falling out of favor with the people inside of Apple who make the Macintosh compilers and write system software. It is very likely that Pascal will continue to be supported by third-party developers, but the Macintosh OS is slowly being converted to C and C++. In making this conversion, the special features of C are being used. As a result Pascal programmers and compiler writers will need to think in C and make the appropriate conversions of the data and function calls to connect the C conventions with those of Pascal.

Religious or not, the Macintosh OS is being written in C and C++ and to make sure that your code is more compatible with our system, we recommend that you learn to love C. If you make the investment now you are assured the easiest transition possible to new platforms. Besides, it is also very hard to make hand tuned assembly which is better than code produced by a good optimizing C compiler, especially on RISC type machines.

You should take full advantage of the features ANSI C provides. For example, you should turn on compiler options to require prototypes and make sure that all your own functions have

prototypes. Be aware that use of "old style" function definitions in MPW C will defeat prototype checking. For example, the following definition of `DoEvent` allows any 4 byte parameter to be passed as the `evtPtr` parameter:

```
void DoEvent (evtPtr)
EventRecord *evtPtr;
{
        .
        .
        .
}
```

Do not assume the C compiler understands Pascal calling conventions. In particular, do not assume the C compiler will automatically pass toolbox structures larger than 4 bytes by reference; do it yourself.

Never use the type `int`. Purists may argue that proper use of `int` gives you the most portable code. But the Macintosh Toolbox is pretty rigid in its use of 16-bit and 32-bit values and experience shows use of `int` just leads to trouble. Use `short` and `long` instead. If you are uncomfortable with `short` and `long`, create your own typedefs such as `int16` and `int32` so you can alter them for different compilers.

Using direct functions in MPW C or inline assembly in Think C is often unavoidable with the current Toolbox, but it is not portable. You should isolate and conditionalize such code. If you write assembly routines for performance reasons, considering writing a C version, for portability, at the same time you write the assembly version.

2) Align Data Structures

The Motorola 68K microprocessors have always been very tolerant of misaligned data structures, but modern, cached computer architectures don't like having to support misaligned data structures. Chances are that the microprocessors which Apple uses in its CPUs will continue to support misaligned data structures, but you will probably find that applications will run considerably faster if the data structures are aligned. This is already the case in the 68040, and it will become more and more important in the future. So if your structure declarations look something like:

```
struct MatchRec {
    unsigned short red;             // 16 bit variable
    unsigned short green;
    unsigned short blue;
    long matchData;                 // 32 bit variable
};
```

Then you may want to change them to look more like:

```
struct MatchRec {
    long matchData;                 // 32 bit variable
    unsigned short red;             // 16 bit variable
    unsigned short green;
    unsigned short blue;
};
```

The problem with the first example is that the long field `matchData` is not aligned on a 32 bit boundary. The third short field, `blue`, offsets the long field from the 32-bit boundary by 16-bits.

3) Don't Depend on 68K Runtime Model (Stacks, A5, Segmentation...)

The 68K runtime model contains many features which are extremely machine dependent (such as A5 worlds) or don't make sense in a modern O/S (i.e. segmentation). Such features should continue to work in a 68K environment, but when you port code to other platforms, assumptions you make about the runtime environment may no longer be valid.

Beware of the assumptions made in the following areas:

• A5 world. This provides two features: application global data and function references, and access to QuickDraw globals. Similar functionality will be provided in other runtime environments, but the method of access will be different.

• Register conventions. A5 is a specific example of a dependency on register conventions. Other examples can be found, such as depending on return values in D0, or A7 being the stack. Beware of similar dependencies on the 68K register model. It will undoubtedly be different on other platforms.

• Calling conventions. Besides emphasis on C calling conventions, different runtime environments are likely to have idiosyncratic calling conventions. Beware of assumptions based on return values, or parameter ordering, location, size or alignment.

• Stack structure. Different runtime environments will use the stack in different ways. Don't assume you know the layout of stack frames, or indeed, even the layout within a stack frame.

• Segmentation. Segmentation will certainly be different, or even nonexistent on future platforms. In most cases this will be a simple and welcome change. For example, you shouldn't have to change your code because #pragma segment directives will simply be ignored if they are not appropriate. On the other hand, segmentation involves fine tuning the memory usage of your application. You may need to rethink your memory strategy in the absence of segmentation. Also, beware of dependencies on other aspects of segmentation, such as the Segment Loader.

• Toolbox dispatching. The current toolbox dispatching mechanism (A-traps) is very 68K dependent and will certainly be different on other platforms. Trap patching is OK, but don't try to short circuit established mechanisms. Don't assume you know the format of the trap dispatch table; use `GetTrapAddress` and `SetTrapAddress` instead. (See also (11)).

4) Isolate and Minimize use of Low-Memory

For as many years as the Macintosh has been shipping there have been applications which have depended on direct access to low-memory globals. Apple has said for a long time that developers must not depend on these global variables, but unfortunately there are many cases where applications MUST access these variables to function. Shared low memory is one area

we must wean developers from before we can provide modern features like protected address spaces and preemptive multitasking.

Low memory usage falls into three categories:

• Documented low memory globals. These are the safest to use as they (or equivalent functionality) will continue to be available. For example, many applications using Standard File depend on the low memory globals `CurDirStore` and `SFSaveDisk`. We can't just wish those away.

• Hardware dependent low memory. Some low memory is specific to the 68K, such as exception and interrupt vectors. Dependence on these locations is bad for two reasons. First, because it will be different on future platforms, and second, because it implies supervisor level access, which may not be allowed in the future. Be very careful about depending on this kind of access. See also (8).

• Undocumented low memory globals. A lot of low memory is used by the system and has never been documented. Yet, applications persist in mucking around in them. This has always been dangerous and unsupported. Apple makes no guarantee that these globals or equivalent functionality will be available in the future.

Because many applications depend on low memory, we can't just pull the rug from under you because every application would break. So, what's a developer to do? *Isolate and minimize your dependence on low memory.*

If accessor functions exist, you should use them. For example, `GetMBarHeight()` returns the same information as the low memory global `MBarHeight`.

Eventually, Apple will provide a new API which will include accessor functions for documented low memory globals and you should use those when available. In the meantime, you might consider using macros to do the same thing. For example, to access `CurDirStore` you might use the C macros:

```
#define GetCurDirStore() (*(long *)CurDirStore)
#define SetCurDirStore(dirID) (*(long *)CurDirStore = (long)dirID)
```

This at least lets you isolate dependencies on `CurDirStore` in your source, so when Apple does change the API, you only need to change your code in one place.

5) Isolate and Minimize use of Internal Toolbox Data Structures

If it isn't documented, don't use it. The data structures which are not documented aren't documented because we expect they may change in the future. This means that if your application is dependent on any internal data structures never changing, then it is very likely that your product will break in the future. There are some major changes being considered for the Macintosh OS and any major change will include the internal data structures.

So beware of any undocumented features which you depend on.

6) Don't Intermix use of Data & Code

It will make it easier for the OS to move to a memory protection model, if the code does not access itself. If you write to the code segments in any way, then we will have problems protecting that code. The way to protect the code is to make the code read only and if there is data or code which is being changed in that code, then we either can't write protect the code, or we break your code. If the data is static, then it is just read only anyway and will not be a problem.

7) Isolate Dependencies on 80 bit extended

Different FPUs are going to have a different preferred format. On 68K, extended offers the best precision as well as the best performance and is therefore the "natural" choice. On other platforms, this may not be the case, so you should be prepared to take these differences into account. For most of you this is not a problem, but it does mean there may be differences in the size of fields in data structures and on disk, as well as in the size of parameters passed to functions. You should try to isolate and avoid dependencies on the size of floating point numbers.

One issue which applications using 80 bit extended numbers might have on the 6888x based machines is extended numbers are faster to work with than doubles. The 6888x chips convert all numbers which it works with to extended and then back to whatever they came in as. Since some other hardware does not necessarily support extended numbers, the fast numbers may be doubles, or something else.

For some developers there is an issue with the perception that your applications need very high precision. There are cases where extended number precision is very useful (i.e. 3D and CAD), but it is possible to make a fully 3D graphics app which does not need extended numbers. If you find that you can not live without extended numbers, then you will find that what you will need to live with is slow calculations.

8) Don't Depend on Interrupt Level or Supervisor Mode

Interrupt levels, supervisor mode, and exception handling are all very dependent on the microprocessor which is being used in the computer your code is running on. If your code thinks that it knows how to manipulate the hardware and system in supervisor mode, then it will find that it is wrong on the next generation of microprocessors. If you think your code can alter the interrupt levels, then once again you will be sadly mistaken. If your code changes the exception handlers, then your code is too dependent on the hardware on which it is running and will more than likely break on Apple's future products.

One common violation of supervisor mode is the use of privileged instructions. In practice, because of compatibility reasons, some instructions may be emulated, but you should not rely on that fact.

The following are the only privileged instructions which are likely to be supported in the future. Other privileged instructions will likely cause Illegal Instruction exceptions.

```
•       ORI.W       #<value>,SR         see note 1
•       ANDI.W      #<value>,SR         see note 1
```

```
•       EORI.W          #<value>,SR             see note 1
•       MOVE.W          <ea>,SR                 see notes 1, 2
•       MOVE.W          SR,<ea>                 see note 2
•       FSAVE           <ea>                    see note 2
•       FRESTORE        <ea>                    see note 2
•       RTE                                     see note 1, 3
•       MOVEC.L         <Rn>,CACR               see note 4
•       MOVEC.L         CACR,<Rn>               see note 4
•       CPUSHA          BC                      see note 4
```

Notes:

1 It is not possible to alter the values of either the S bit or the M bit with these instructions. The S and M bits you supply are simply ignored by the emulation of these instructions.

2 It should be possible to support all effective address modes for any instruction which uses an effective address operand providing that mode is legal for this instruction. Use of illegal addressing modes results in an Illegal Instruction exception.

3 Only normal four word frames are supported by the emulation of RTE. Other frame kinds generate Illegal Instruction exceptions.

4 Use of MOVEC (or CPUSH and CINV on 68040 machines) to control the instruction and data caches is strongly discouraged. Developers should use system software routines such as `FlushInstructionCache()` to accomplish the same thing.

9) 32 Bit Clean Mandatory

Apple has been saying for many years that all applications need to be able to run in a 32-bit address environment. Not only is this important because it allows users to take full advantage of the memory of the machine, but it allows Apple to transition from one memory model (24-bit) to another memory model (32-bit). Supporting two different memory models has allowed Apple to maintain compatibility, but the cost has been a lot of extra code in the ROM and some performance penalty. So now we're preparing to take the next step – 32-bit only addressing. So, be forewarned.

Most developers have by now removed any dependency on a 24-bit environment, but some developers have cheated a bit. Rather than making use of the upper 8-bits of an address they use the upper bit, on the assumption (valid for the most part) that everything happens in the lower 2 gigabytes of the 4 gigabyte address space. In other words, they are only 31-bit clean. This is very bad as future operating systems will take full advantage of the 4 gigabyte range of addresses.

So, the message is, not only will we be 32-bit only, but we really mean 32-bit.

10) Keep hands off all Hardware Registers

Apple does not support 3rd party products accessing any of the hardware registers. The reason for this is that we know that the hardware will change and when it does change, we do not

wish to have a large number of applications breaking. If we did allow products to access the hardware registers directly, then we would never be able to allow the Macintosh hardware to evolve as technology evolves. If this were the case, the Macintosh would have been a dead product by now.

We know that there are products which do depend on our hardware never changing and these products have had many compatibility problems. We know that these problems will only continue to get worse with time. The hardware will be going through a great deal of changes in the coming years and these changes will not be able to include compatibility with any product which accesses the hardware registers directly.

We don't want to see any of these products break, but the only thing we can do is to encourage developers not to depend on the Hardware - So Don't.

11) Don't directly patch the ROM

Do not access the trap tables directly. Use `SetTrapAddress` and `GetTrapAddress` to guarantee future compatibility. The next major ROM will use a vectorization method for patching. Vectorization is definitely going to affect the trap tables in the future. In the near future, we may even bypass the trap table and make `Set/GetTrapAddress` use the vector locations. Also for RISC, we may implement two copies of the tables, one being for native code.

Below is a sample of the correct way to patch a trap. This code is from the 7.0 sample for making an INIT/cdev.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;MACRO
;       ChangeTrap      &trap, &type, &newAddress, &oldAddressStore
;
;       Macro used to change the trap address and optionally save the old
;       routine's address. You pass in the trap number of the trap you want
;       to patch, the type of the trap (newTool or newOS), the address of the
;       routine to store in the trap table, and a pointer to a memory location
;       that will hold the old address. If &oldAddressStore is some value other
;       than NIL, this macro will get the old trap's address and save it there.
;
;       NOTE:   This macro translates &newAddress and &oldAddressStore into
;               their new locations. To do this, it relies on A1 pointing to
;               the block in the system heap that holds the patch, and for
;               FirstResByte to be defined.
;
;Input:
;       A1: address of patch code in system heap
;
;Output:
;       oldAddressStore: address of old trap routine
;       D0, A0 are destroyed.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        MACRO
        ChangeTrap      &trap,&type,&newAddress,&oldAddressStore
```

```
	IF (&oldAddressStore ≠ 'NIL') THEN

		move.w #&trap,D0
		_GetTrapAddress ,&type
		move.l A0,&oldAddressStore-FirstResByte(A1)

	ENDIF

		move.w #&trap,D0
		lea		&newAddress-FirstResByte(A1),A0
		_SetTrapAddress ,&type

	ENDM
```

An alternate piece of sample code which works very well for Think C follows:

```
#include <SetUpA4.h>


/*
 *generic code patch loader
 *MacDTS/pvh
 *©1989-90 Apple Computer, Inc.
 *Copies CODE resource into system heap.  puts original trap address
 *at head of code in case you need it later.  This is THINK C specific of course.
*/

#define GetNextEvent_trap    0xA970

/*
 *this is need because the Quickdraw global thePort is not defined at INIT time
*/
typedef struct myQDGlobalsDef {
	char		qdPrivates[76];
	long		randSeed;
	BitMap		screenBits;
	Cursor		arrow;
	Pattern		dkGray;
	Pattern		ltGray;
	Pattern		gray;
	Pattern		black;
	Pattern		white;
	GrafPtr		thePort;
	} myQDGlobalsDef;

/* OS traps start with A0, Tool with A8 or AA. */
short GetTrapType(short theTrap)
{
	if((theTrap & 0x0800) == 0)		/* per D.A */
		return (OSTrap);
	else
		return (ToolTrap);
}

/*
 *The INIT
*/
void main()
{
	Handle		h;
	long		size;
	Ptr		codeSpot;
```

```
        KeyMap                  theKeyMap;
        myQDGlobalsDef          myQDGlobals;
        long                    trapAddr;
        long                    *blah;

        RememberA0();
        SetUpA4();

        InitGraf(&myQDGlobals.thePort);
        GetKeys(&theKeyMap);

        /* don't install if shift key is down */
        if(theKeyMap.Key[1] != 1L)    {
                h = GetResource('CODE', 12);        /* our patch code resource */

                if (h != 0L) {
                        size = SizeResource(h);

                        /* size of our resource + 4 for saved trap address */
                        codeSpot = NewPtrSys(size+4L);
                        HLock(h);       /* lock the resource handle just because */
                        /* move in the CODE resource into the SYS heap */
                        BlockMove(*h, codeSpot+4L, size);
                        HUnlock(h);     /* unlock it */

                        /* get the 'current' trap address */
                        trapAddr = NGetTrapAddress(GetNextEvent_trap,
                                GetTrapType(GetNextEvent_trap));

                        /* this is skanky but move the original trap */
                        blah = (long *) codeSpot;
                        /*address into the topof the new block.  Would */
                        *blah = (long) trapAddr;
                        /* set to the new trap address */
                        NSetTrapAddress(codeSpot + 4L, GetNextEvent_trap,
                                GetTrapType(GetNextEvent_trap));



/*
; …or use this assembler source if you'd rather
asm {
        move.l size, d0         ; size of our resource
        add.l  #4, d0           ; add 4 to size for place holder for real trap address
        _NewPtr         SYS     ; create block in system heap
        move.l a0, codeSpot     ; spot to save pointer

        move.l h, a0
        _HLock                  ; lock the handle of our code

        move.l size, d0         ; size of our resource
        move.l codeSpot, a1     ; head of block
        adda.l #4, a1           ; we want the first 4 bytes for saving original trap
                                ; address
        move.l h, a0            ; handle to our patch code
        move.l (a0), a0         ; dereference to actual address
        _BlockMove              ; move it in

        move.l h, a0
        _HUnlock                ; unlock the handle

; save the real trap address in 4 byte spot at head of block
        move.w #GetNextEvent_trap, d0
```

```
        _GetTrapAddress
        move.l codeSpot, a1
        movem.l (a0), d1
; set the trap address to our patch, 4 bytes past the block header
; (remember the first 4 bytes our the saved original address)
        move.w #GetNextEvent_trap, d0
        move.l codeSpot, a0
        adda.l #4, a0
        _SetTrapAddress
}
*/

        RestoreA4();
}

Sample patch code for _GetNextEvent using THINK C (there is a little assembly
involved):
This would be compiled in a separate project as CODE resource ID=12

pascal Boolean main(short mask, EventRecord *evt)
{
        long    realTrapAddr ;

        asm {
            movem.l d0/a2-a6,-(sp)
            ; THINK C kindly places pointer to top of CODE resource in a0
            ; we saved the original trap address just ahead of our patch so
            ; let's go get it and save it
            move.l a0, a1
            sub.l  #4, a1
            move.l (a1), realTrapAddr
            }
        /*do the THINK stuff */
        RememberA0();
        SetUpA4();

        SysBeep(1);

        /*do the THINK stuff */
        RestoreA4();
        asm {
            movem.l (sp)+, d3-d7/a2-a6
            move.l realTrapAddr, a0      ; get original trap address
            unlk    a6                   ; unlink
            jmp     (a0)                 ; and jump to it
            }
}
```

12) Don't depend on resources being in the System File

In ta future version of the ROM we plan to put parts of System 7 into the ROM. This means that if you expect to see a resource in the system file, then you may find yourself in trouble. Some of the System 7 resources and packages will soon reside in the ROM.

13) ROM version number may not tell you what you want to know

Whatever reason you may have for checking the ROM version, it is unlikely to be a valid reason in the future. It is possible that the IDs will be different for each ROM, but the software in the ROMs will be the same. It is also possible for the same ROM to operate differently depending on what hardware the ROM is on. This means that the IDs will be the same, but the code will not.

As useless as the ROM version information is expected to be, if you still feel that you need this information, use Gestalt to get it.

14) Don't assume ROM size - It will grow

If you have not noticed, the ROM keeps getting bigger and bigger. The ROM for Quadras is 1 meg and it is going to get even bigger with each new version. The next jump is expected to be anywhere from 2 to 4 meg in size.

15) SCSI

There are some things which you should already know about SCSI, but it is important to reiterate some potential problems which you may have already experienced and probably will continue to see in the future.

SCSIStat:

Although the `SCSIStat` routine has been provided in the SCSI Manager ever since the Mac Plus it is a mostly useless routine. Although it does provide some information on all the Macintosh computer until the Quadras, this information has encouraged some developers to design products which did not follow the SCSI specification. If your device is a true SCSI device, then the device and its driver should never care what the state the SCSI bus is in. Some developers found on the Quadra computers that we changed the hardware which we were using to improve the performance and lead Apple's hardware into the future of SCSI. This change caused several developers to have problems, in part because the new hardware provided no way for the call `SCSIStat` to perform with anything which resembled accuracy or reliability.

Protocol:

Another problem which some developers had with SCSI on the Quadras was related to their not following the SCSI specification on protocol. The Target is in control of the bus and all the driver does is call the appropriate routines in the correct order. If you depend on these routines being synchronous and providing feedback as they are being called, then your product had problems on the Quadras. On the Quadras, the SCSI operations are queued up until the

SCSICmd is sent. At this time the SCSI transaction is done in one big operation. There are no errors reported along the way, so the drivers must operate on the assumption that the SCSI Manager knows what it is doing and that the hardware knows what it is doing.

It is very important to follow the SCSI specification.

Patches:

If your products patch any part of the SCSI Manager at this time, be aware that the SCSI Manager is soon going to go through a major overhaul. Your patches may cause problems with the new SCSI Manager, so keep an eye out for all documentation which Apple generates on the SCSI Manager. If you feel that you may need some advanced warning about the new SCSI Manager, then we highly recommend that you contact Apple Evangelism and make sure they understand that you feel you might have problems with the new SCSI Manager and that you would like to receive any documentation they have when it is available.

16) VIAs

There are some 3rd party products which like to interrogate the VIAs for information about ADB activity or get high resolution timing from the VIAs. There are other products which use the VIAs to change the interrupt levels of the hardware. Any products which depend on the VIAs in any way are going to have a great deal of trouble in the future as Apple hardware gradually depends less and less on having VIAs. You may have started to see some of the Macintosh either eliminating one of the VIAs or emulating the VIAs in a big ASIC. As the hardware continues to evolve, these VIAs will become even less available.

17) Do The Right Thing

"WHY ASK WHY", "JUST DO IT"

If you need to find out if something is available for your use, then use Gestalt to find out. If Gestalt does not tell you what you need to know, then you may be looking for something which you should not be worrying about. There are some cases where this is not true, but in most cases it is.

It is also important that you look for the exact information which you need to know. If you want to know if there is an FPU, then check for that – nothing else will tell you the truth.

Last, but not least – **when in doubt ASK!**