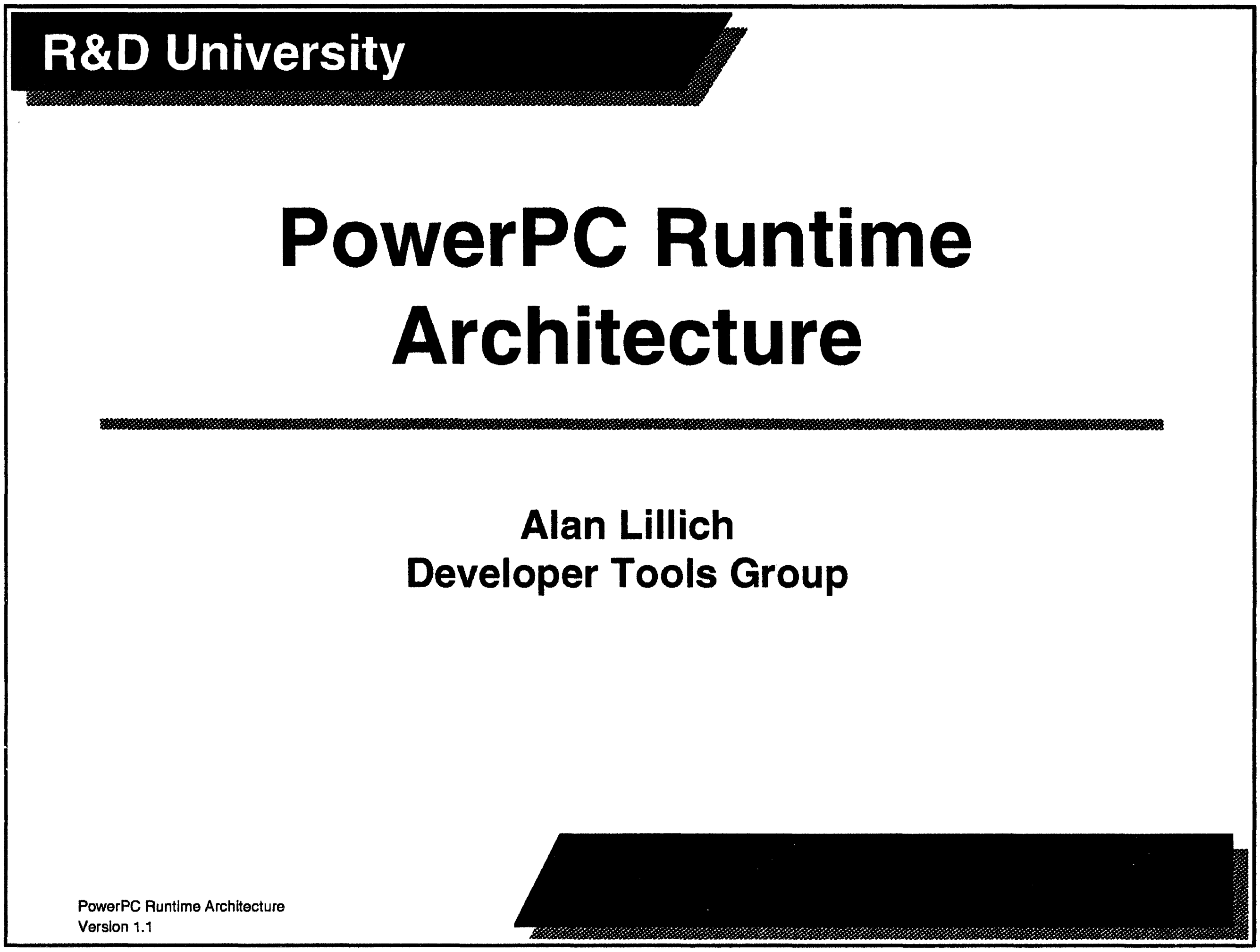
R & D University

PowerPC Runtime Architecture

R&D University

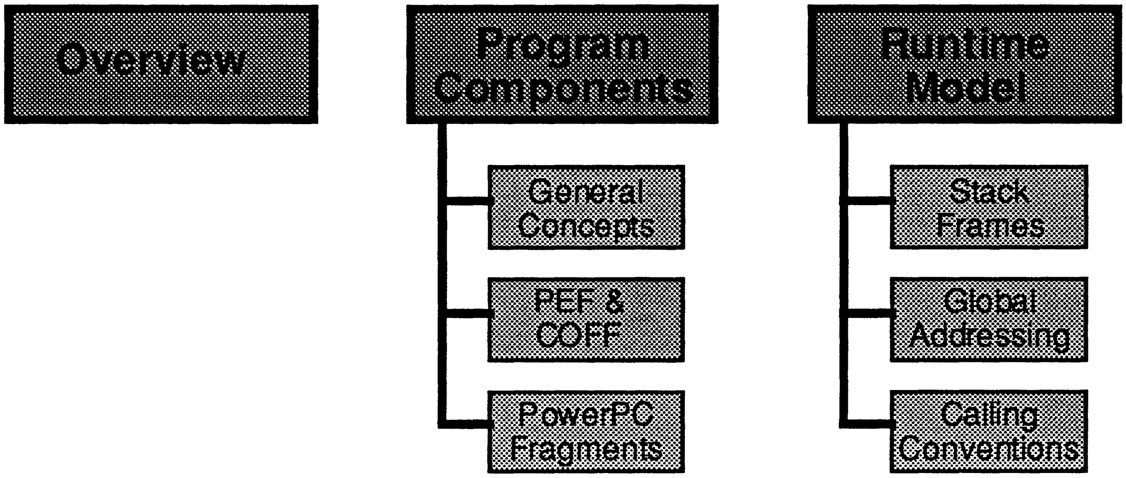


R&D University

PowerPC Runtime Architecture

Alan Lillich
Developer Tools Group

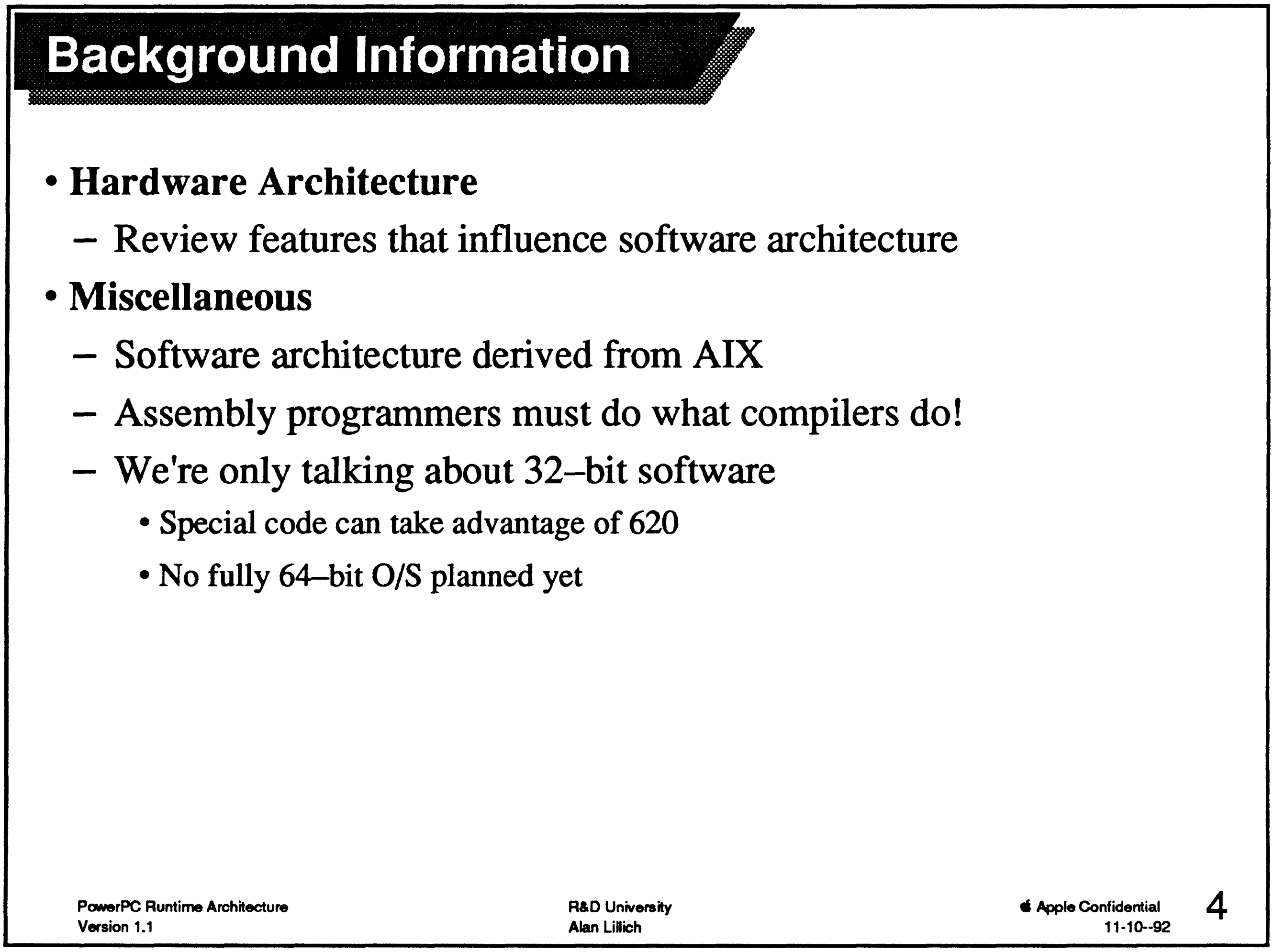
Day 1 Content



Schedule for Today

First Day of Course

- Background Information
- Program Components
- Global Addressing
- Stack Frames
- Calling Conventions



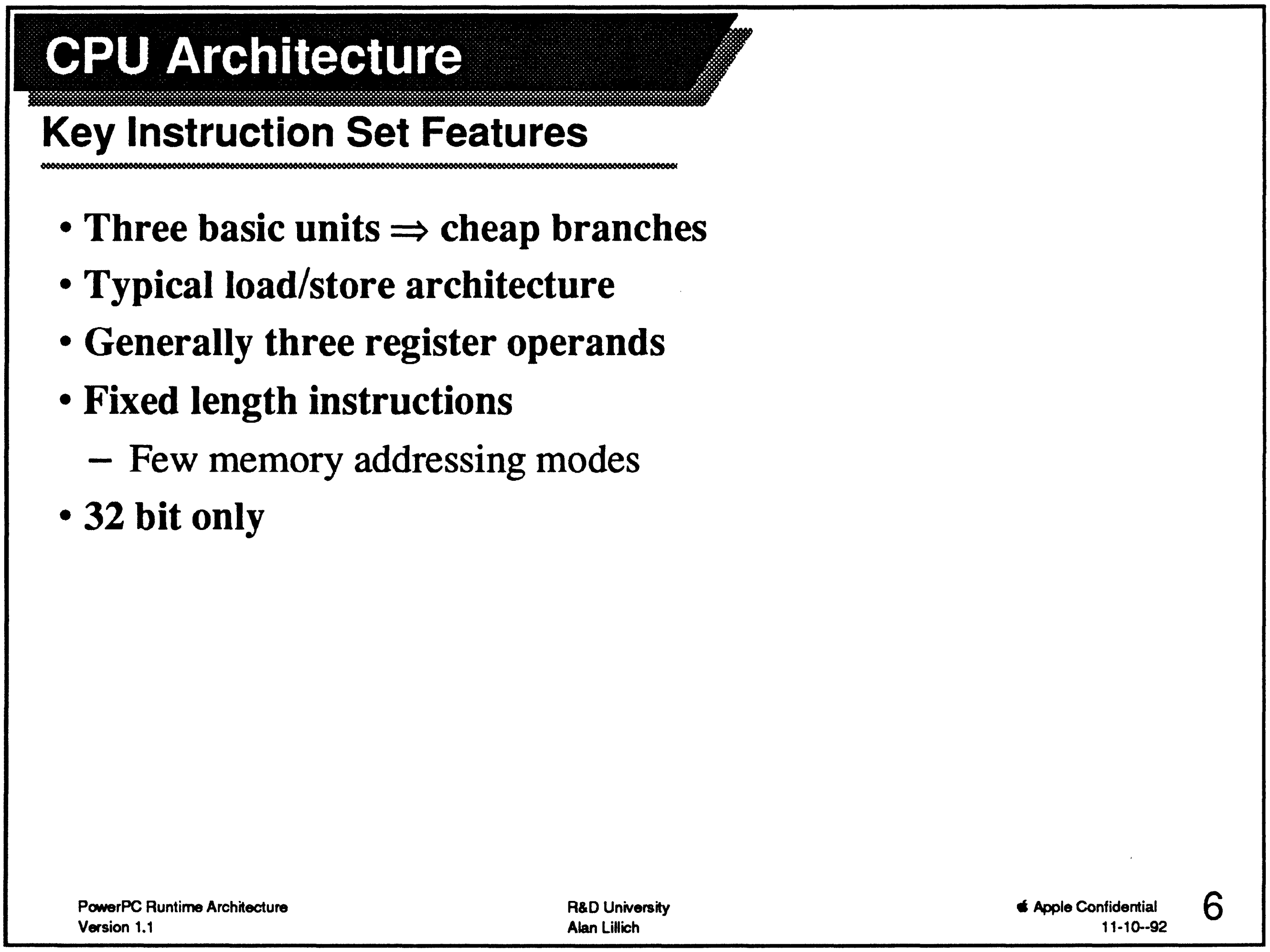
Background Information

- **Hardware Architecture**
  - Review features that influence software architecture
- **Miscellaneous**
  - Software architecture derived from AIX
  - Assembly programmers must do what compilers do!
  - We're only talking about 32–bit software
    - Special code can take advantage of 620
    - No fully 64–bit O/S planned yet

Background Information

Definitions

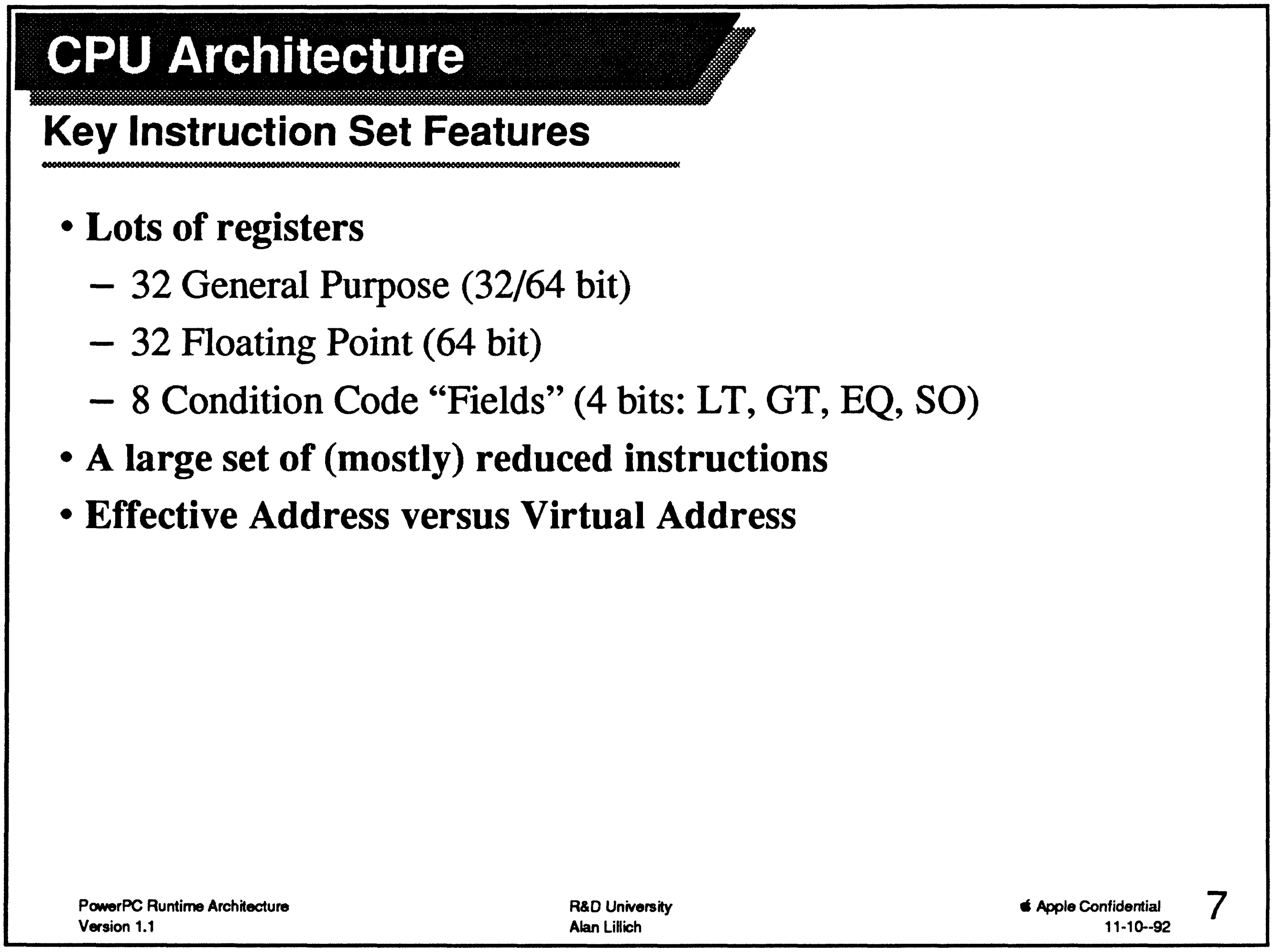
- **Effective Address:** 32 bits
  - The "register size" addresses used by PowerPC software.
- **Virtual Address:** 80 bits
  - A "large" address used by PowerPC hardware during address translation.
- **Volatile Register:**
  - A register whose contents need not be the same on return from a call as before.



CPU Architecture

Key Instruction Set Features

- Three basic units ⇒ cheap branches
- Typical load/store architecture
- Generally three register operands
- Fixed length instructions
  - Few memory addressing modes
- 32 bit only



CPU Architecture

Key Instruction Set Features

- **Lots of registers**
  - 32 General Purpose (32/64 bit)
  - 32 Floating Point (64 bit)
  - 8 Condition Code “Fields” (4 bits: LT, GT, EQ, SO)
- **A large set of (mostly) reduced instructions**
- **Effective Address versus Virtual Address**

CPU Architecture

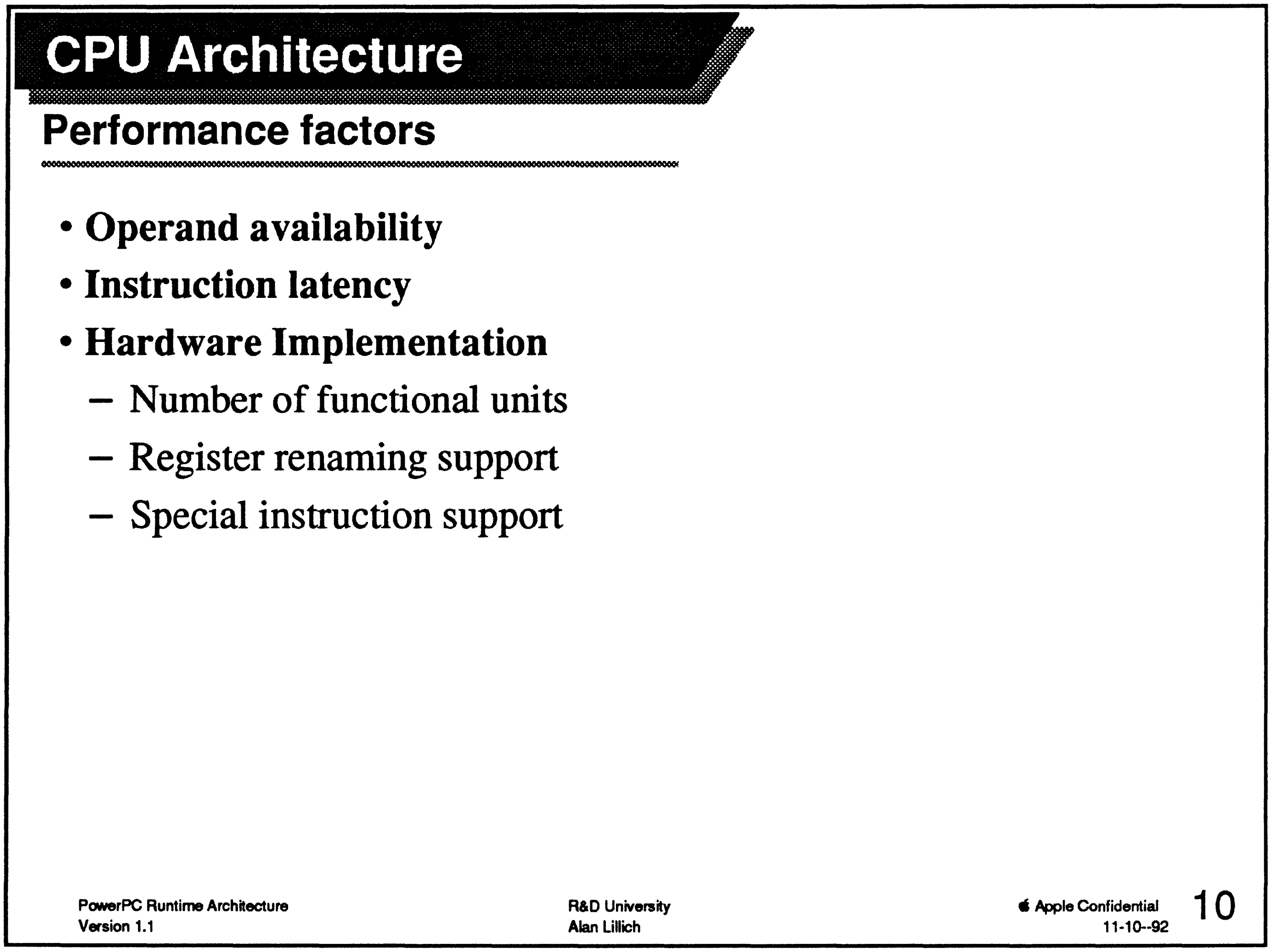
Instruction Addressing Modes

- **Unconditional Branches** instructions word offsets
  - PC relative, 24–bit immediate displacement (±32MB) bytes
- **Conditional Branches**
  - PC relative, 14–bit immediate displacement (±32KB)
  - Indirect through Link or Count Register
- **Displacements**
  - Are signed and in words (instructions), not bytes
  - May be taken as absolute address (lowest & highest 32xB)
- **Calls save return address in Link Register**

CPU Architecture

Data Addressing Modes

- **Base + Displacement Addressing Mode (D–form)**
  - Base: Any GPR other than R0 (R0 ⇒ zero for base)
    May be updated with Effective Address
  - Disp.: 16-bit signed immediate value (±32KB)
- **Base + Index Addressing Mode (X–form)**
  - Base: Same as D–form
  - Index: Any GPR (including R0)
- **Large displacements need multiple instructions**
  - Construct absolute address
  - Construct 32-bit offset
  - Load pointer



CPU Architecture

Performance factors

- **Operand availability**
- **Instruction latency**
- **Hardware Implementation**
  - Number of functional units
  - Register renaming support
  - Special instruction support

Register Conventions

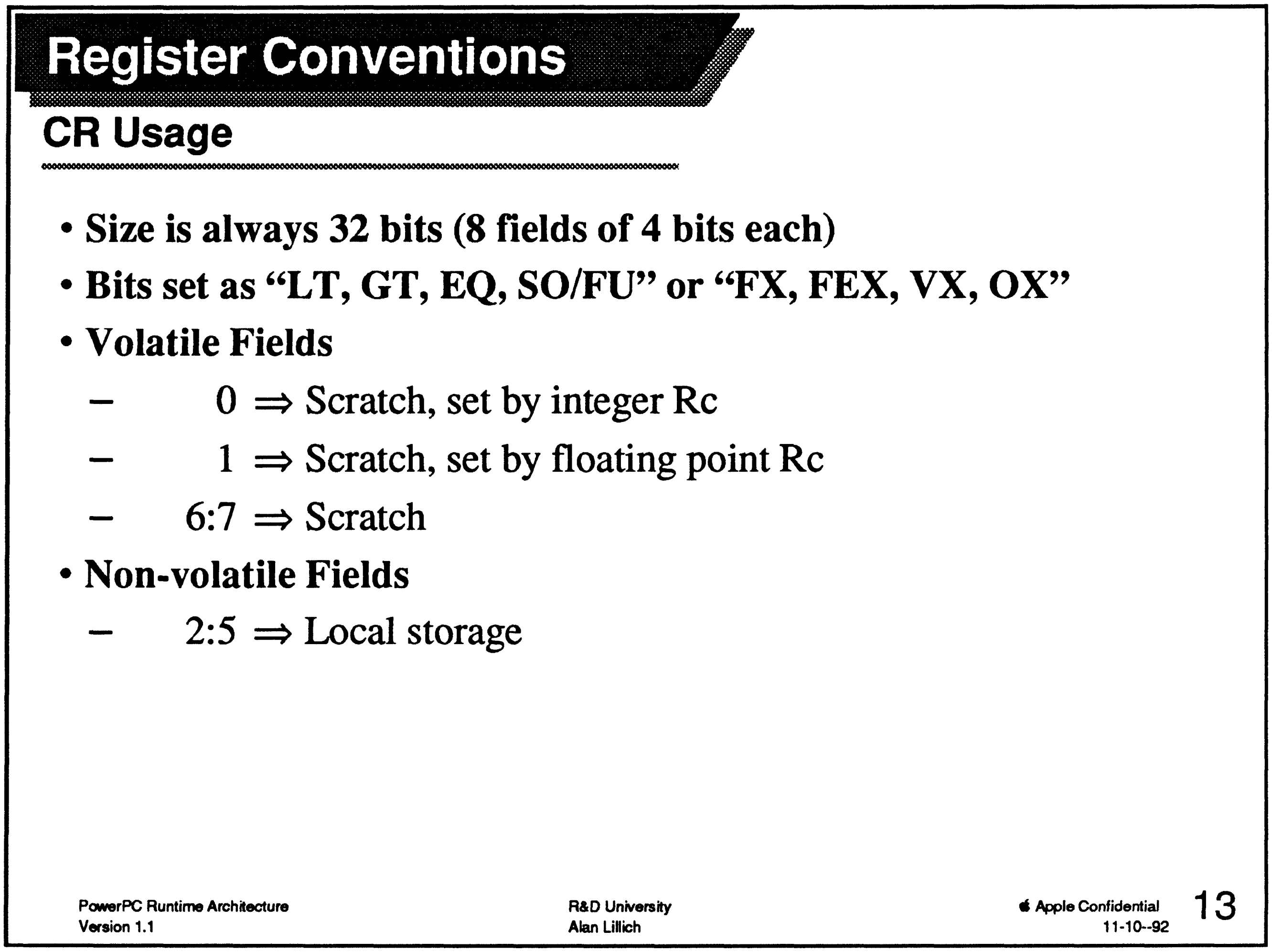
GPR Usage

- **Size is 32 bits or 64 bits**
- **Volatile Registers (may be clobbered by calls)**
  - 0 ⇒ Scratch, glue, prologues, and epilogues
  - 3:10 ⇒ Scratch, "integer" and composite parameters
  - 11:12 ⇒ Scratch, glue, prologues, and epilogues
- **Nonvolatile Registers (preserved by calls)**
  - 1 ⇒ Stack pointer
  - 2 ⇒ TOC pointer
  - 13:31 ⇒ Local storage

Register Conventions

FPR Usage

- **Size is always 64 bits**
- **Volatile Registers**
  - 0 ⇒ Scratch
  - 1:13 ⇒ Scratch, floating point parameters
- **Non-volatile Registers**
  - 14:31 ⇒ Local storage



Register Conventions

CR Usage

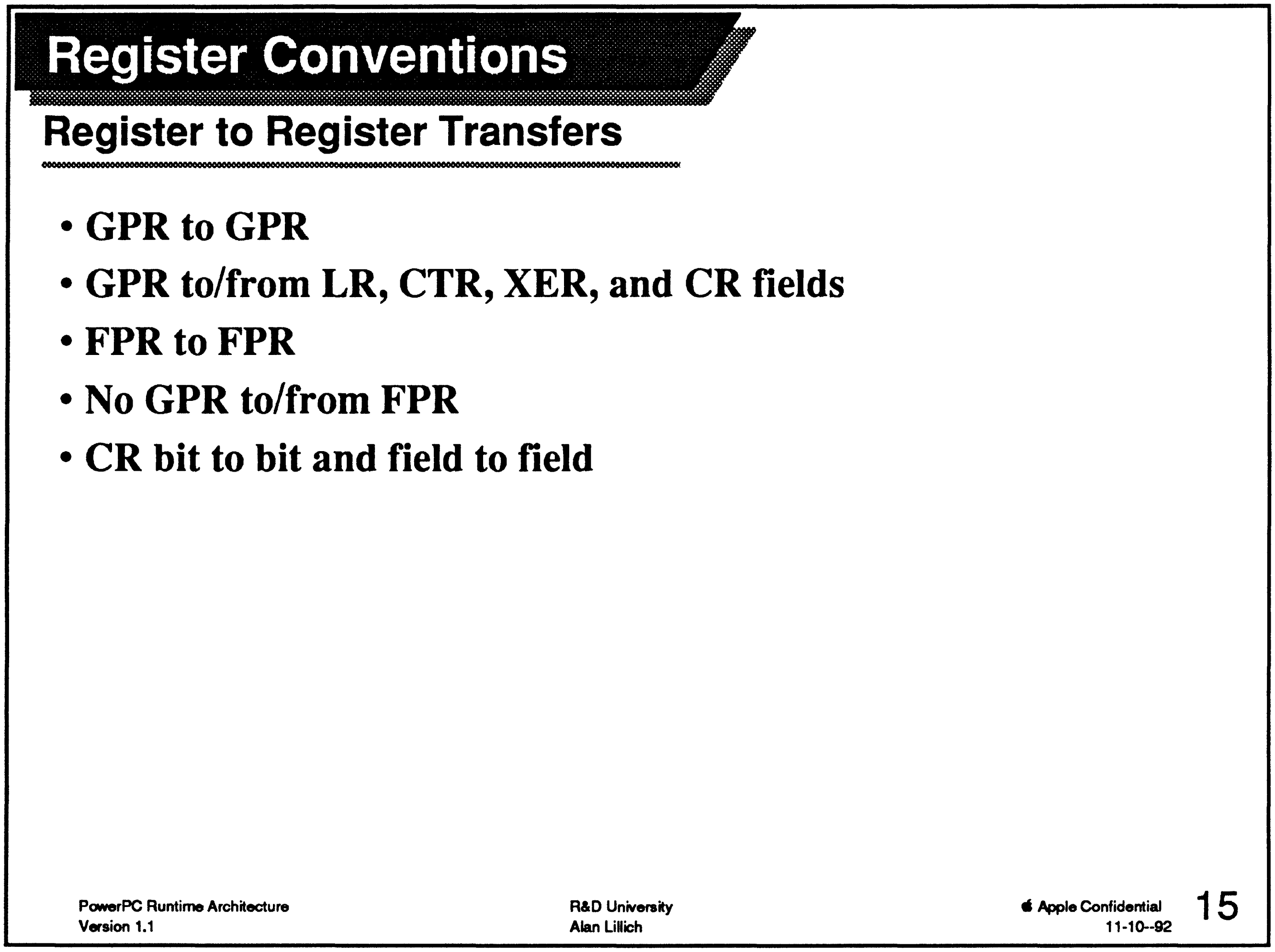
- **Size is always 32 bits (8 fields of 4 bits each)**
- **Bits set as “LT, GT, EQ, SO/FU” or “FX, FEX, VX, OX”**
- **Volatile Fields**
  - 0 ⇒ Scratch, set by integer Rc
  - 1 ⇒ Scratch, set by floating point Rc
  - 6:7 ⇒ Scratch
- **Non-volatile Fields**
  - 2:5 ⇒ Local storage

Register Conventions

Special Register Usage

Name	Size	Volatile?	Usage
LR	32/64	Yes	Routine call/return
CTR	32/64	Yes	Loop counts, computed branches
XER	32	Yes	Integer status and byte counts
FPSCR	32	Mixed	Float status and control

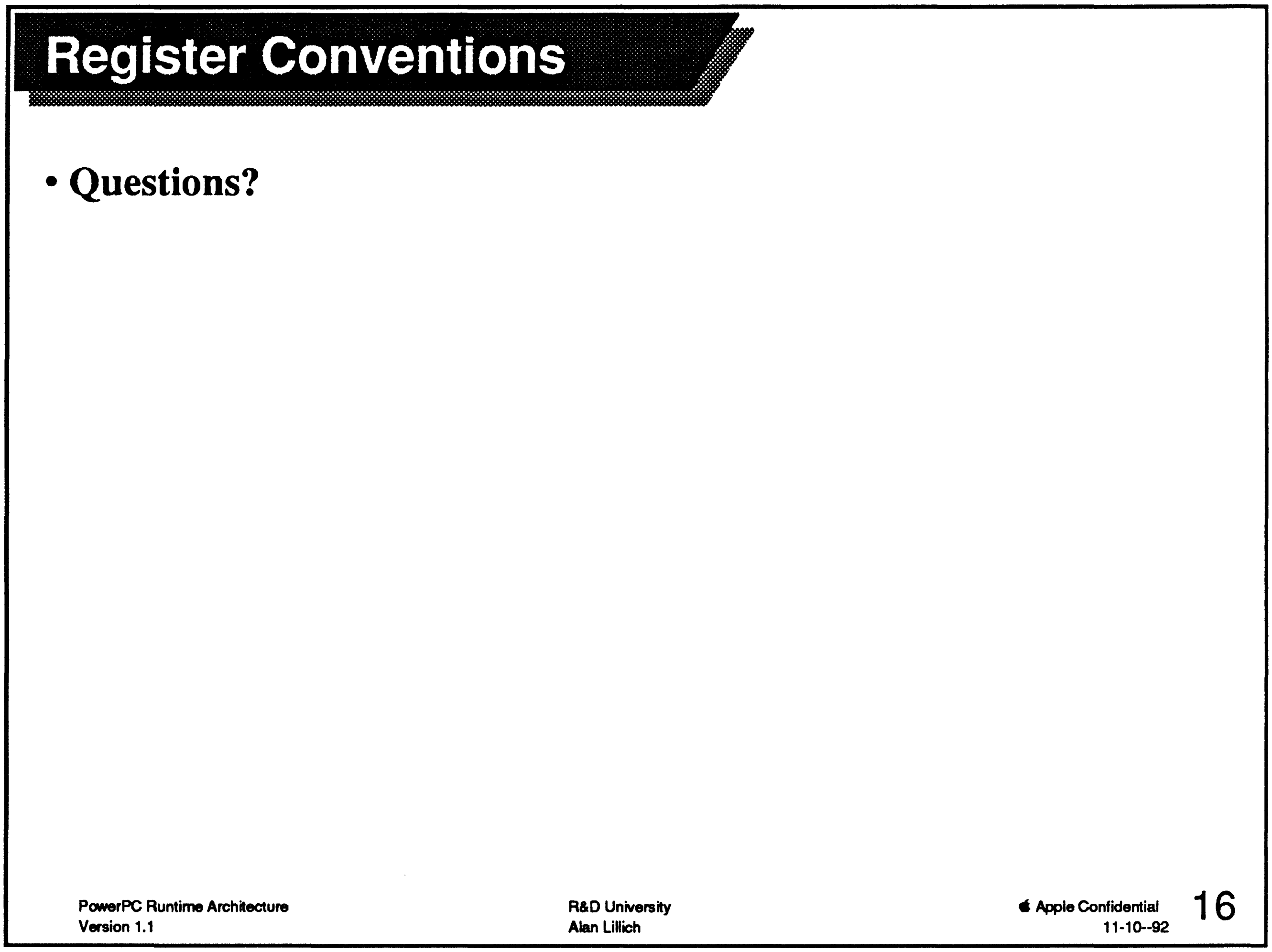
FPSCR status bits are volatile, control bits are nonvolati



Register Conventions

Register to Register Transfers

- GPR to GPR
- GPR to/from LR, CTR, XER, and CR fields
- FPR to FPR
- No GPR to/from FPR
- CR bit to bit and field to field



Register Conventions

- **Questions?**

Program Components

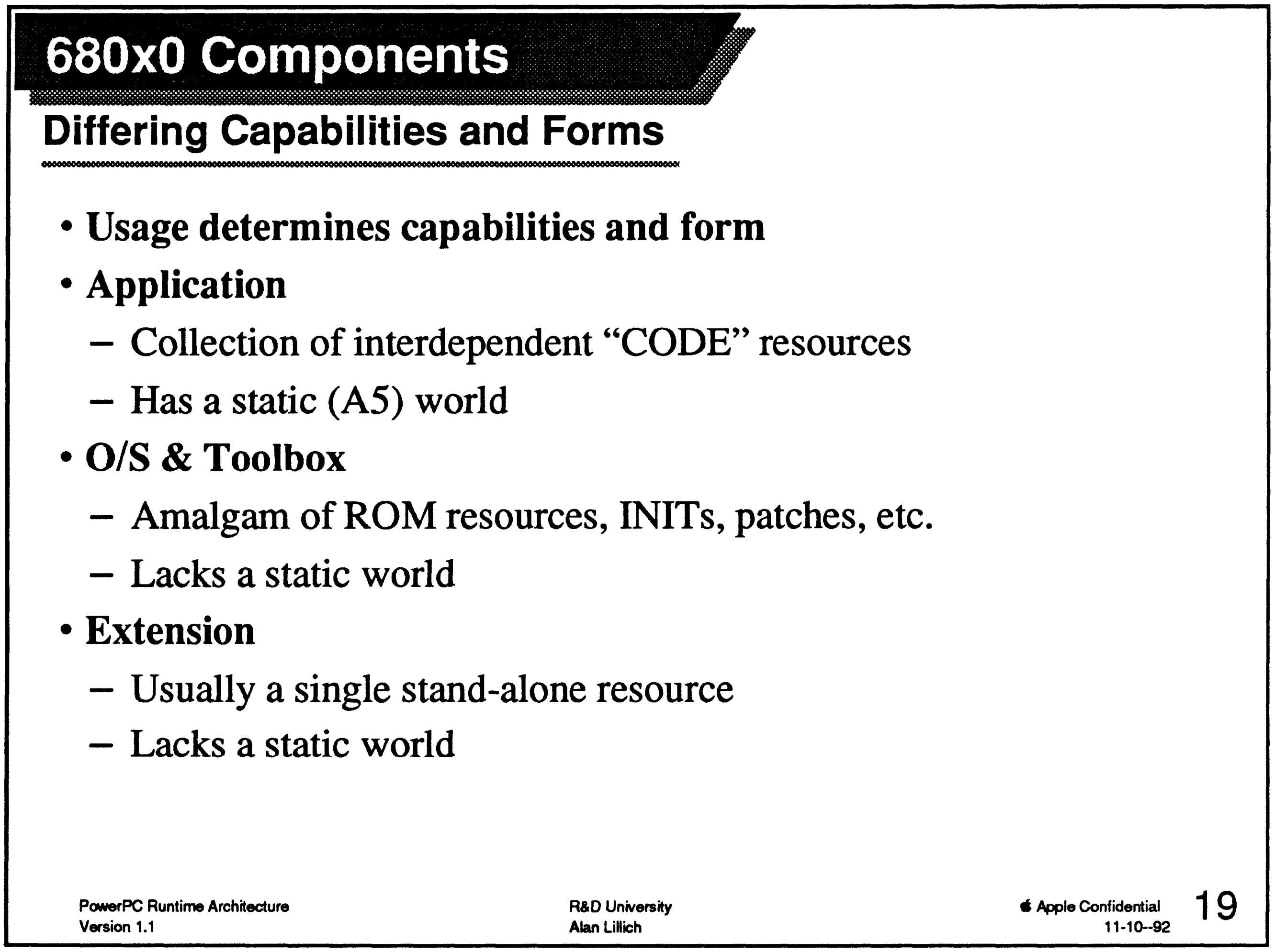
Basic physical building blocks

- **Differentiated by:**
  - Usage
    - Applications
    - O/S and Toolbox
    - “Extensions”
  - Capabilities
    - Supports static data (“A5 world”)
    - Size limitations
  - Storage location and form
    - In resource or data fork
    - Type and number of resources
    - Internal format of storage

Program Components

Definitions

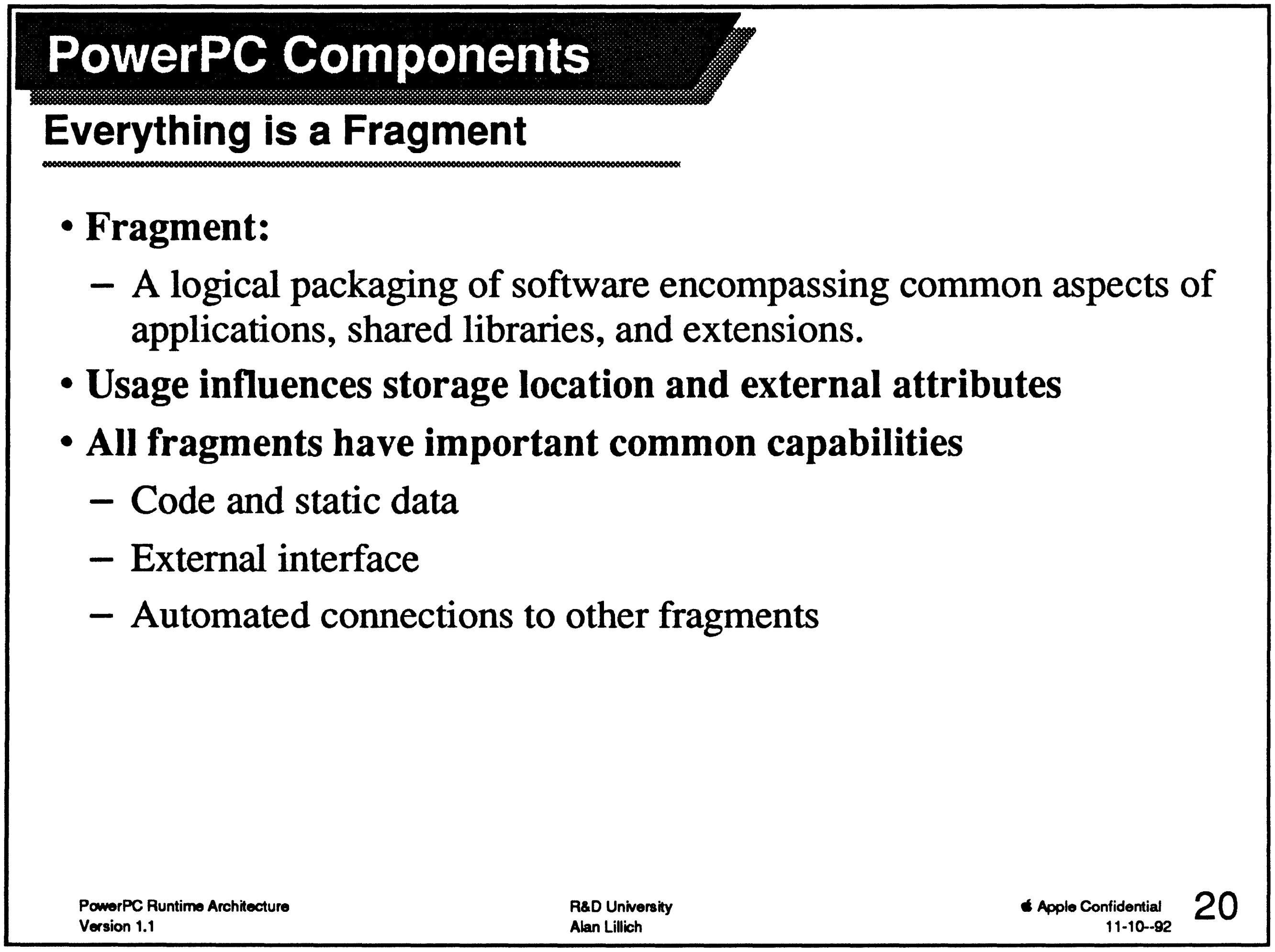
- **Application:**
  - Software launched by the Finder to process documents.
- **Shared Library:**
  - Software used at link time to resolve external symbols and again (automatically) at runtime to provide implementation of those resolved symbols.
- **Extension:**
  - Software that is neither an application nor a shared library. Activated ~~manually~~ by program at runtime. Examples XCMD, LDEF, DRVR, INIT, CDEV.



680x0 Components

Differing Capabilities and Forms

- **Usage determines capabilities and form**
- **Application**
  - Collection of interdependent “CODE” resources
  - Has a static (A5) world
- **O/S & Toolbox**
  - Amalgam of ROM resources, INITs, patches, etc.
  - Lacks a static world
- **Extension**
  - Usually a single stand-alone resource
  - Lacks a static world



PowerPC Components

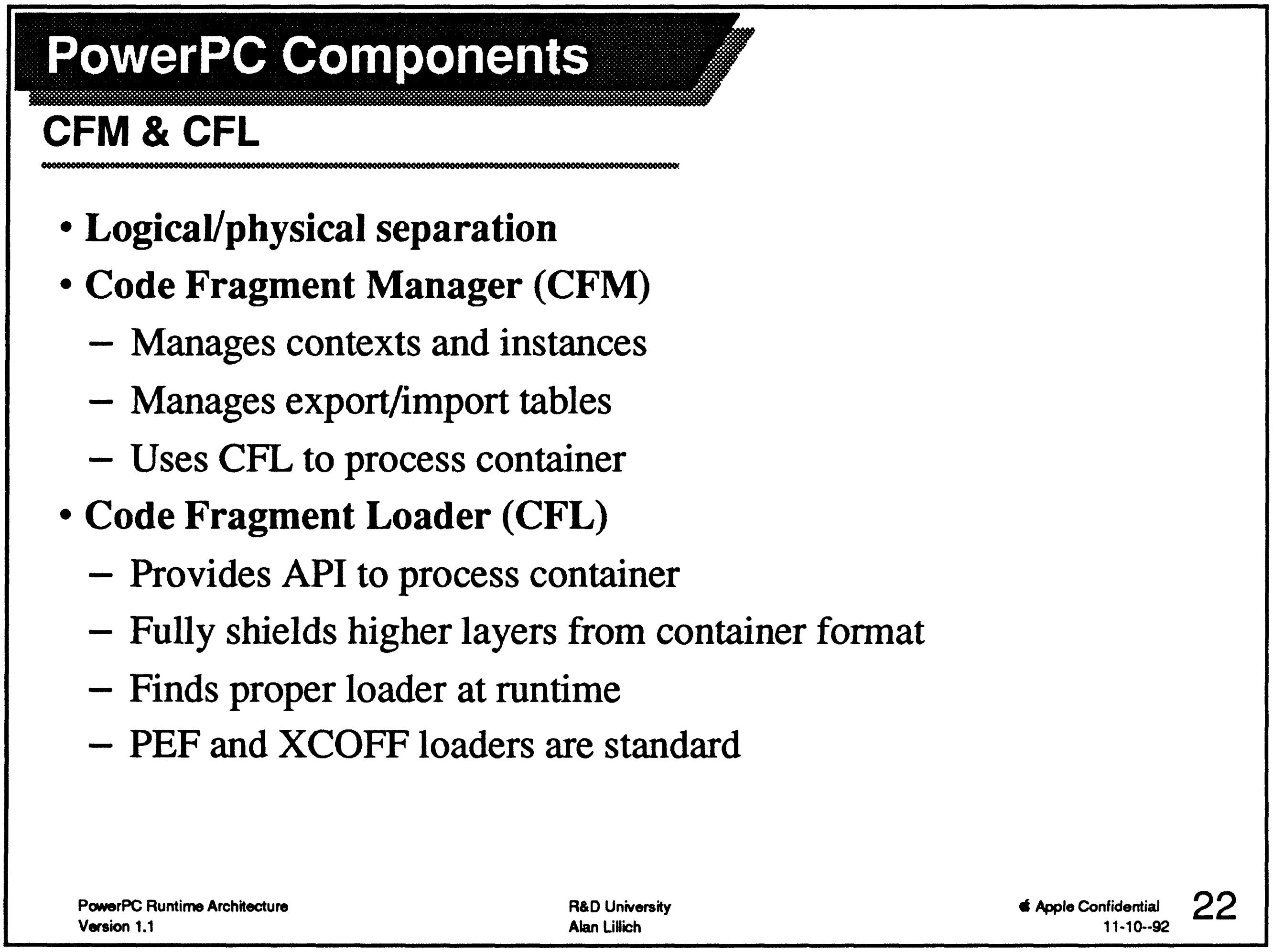
Everything is a Fragment

- **Fragment:**
  - A logical packaging of software encompassing common aspects of applications, shared libraries, and extensions.
- **Usage influences storage location and external attributes**
- **All fragments have important common capabilities**
  - Code and static data
  - External interface
  - Automated connections to other fragments

PowerPC Components

Everything is a Fragment

- **Container:**
  - The physical storage of a fragment.
- **Storage location can be anywhere**
- **Internal storage formats hidden by loader API**
- **Linker creates fragments**
  - One link ⇒ one fragment
  - Fragments are not segments ← not existing Mac segments
- **Section:**
  - A region of memory occupied by part of a loaded fragment



PowerPC Components

CFM & CFL

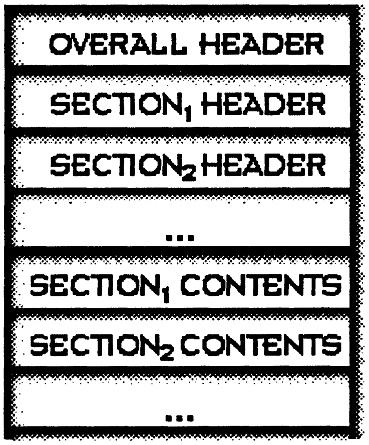
- **Logical/physical separation**
- **Code Fragment Manager (CFM)**
  - Manages contexts and instances
  - Manages export/import tables
  - Uses CFL to process container
- **Code Fragment Loader (CFL)**
  - Provides API to process container
  - Fully shields higher layers from container format
  - Finds proper loader at runtime
  - PEF and XCOFF loaders are standard

SLM will be implemented over CFM on PowerPC

PEF & XCOFF Containers

Similar Organization

- **Single contiguous piece of storage**
- **Headers are of fixed length**
- **Typically three sections:**
  - Code (loaded)
  - Static data (loaded)
  - Loader information (not loaded)
- **Loader information describes**
  - Loaded sections
  - Exports & Imports
  - Runtime relocations



OVERALL HEADER

SECTION$_1$ HEADER

SECTION$_2$ HEADER

...

SECTION$_1$ CONTENTS

SECTION$_2$ CONTENTS

...

PEF & XCOFF Containers

Differences in Usage

- **XCOFF headers and sections tied to UNIX memory model**
- **PEF truly supports multiple code and data sections**
- **PEF is defined only as an executable format**
  - No defined sections for linker relocations, debugging, etc.
  - Could be extended, but there are no plans to do so
- **Tool usage**
  - IBM's linker only generates XCOFF
  - Apple's linker initially generates XCOFF
  - A conversion tool reads XCOFF and writes PEF

PEF & XCOFF Containers

Differences in Features

- **PEF has smaller headers**
- **PEF supports ...**
  - ... data sharing attributes
  - ... packing for initialized data
  - ... bidirectional version checks for imports
  - ... initialization and termination routines
- **PEF has a dramatically improved loader section**
  - Tremendous savings for runtime relocations
  - Better organization for export/import tables

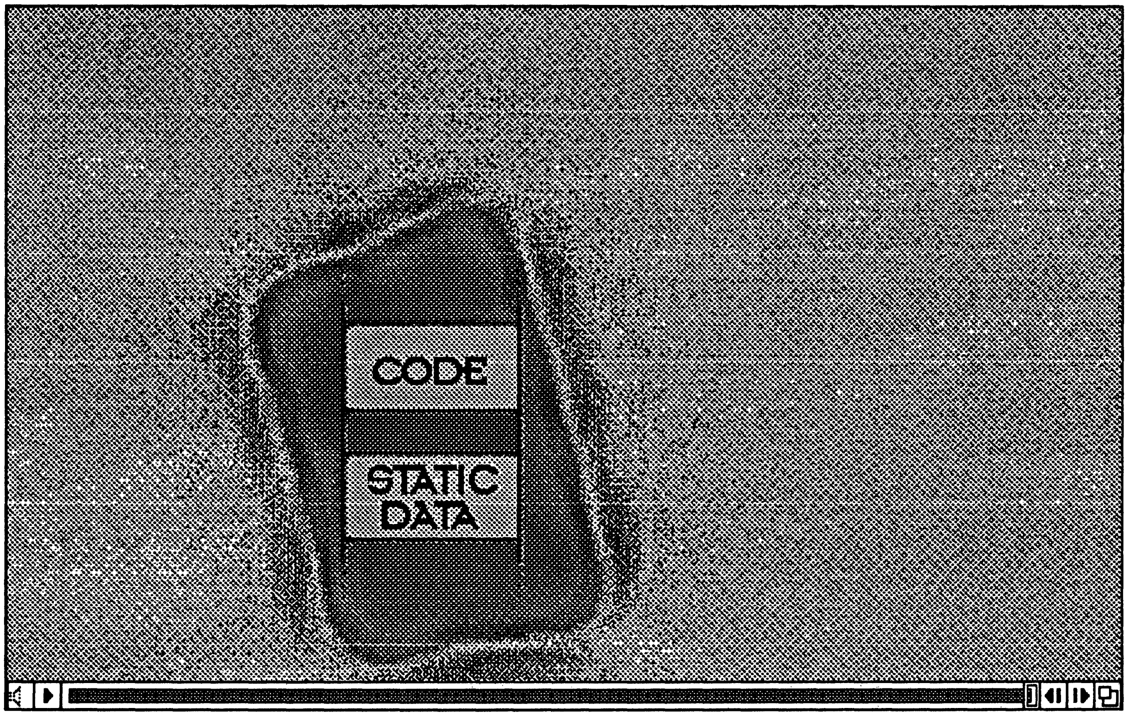
PEF & XCOFF Containers

Loader Section Improvements

- **Runtime relocations (all are 32–bit pointers)**
  - XCOFF uses 12 bytes per relocation
  - PEF uses one 2 byte item for many relocations
  - PEF takes advantage of relocation patterns
    - Linker groups all transition vectors together
    - TOC pointers to imported symbols are often contiguous
    - TOC pointers to own code or data are often contiguous
    - Initialized data like C++ VTables include relocation
- **Export/Import tables**
  - XCOFF merges, PEF organizes by usage
  - PEF contains hash tables for CFM

PowerPC Components

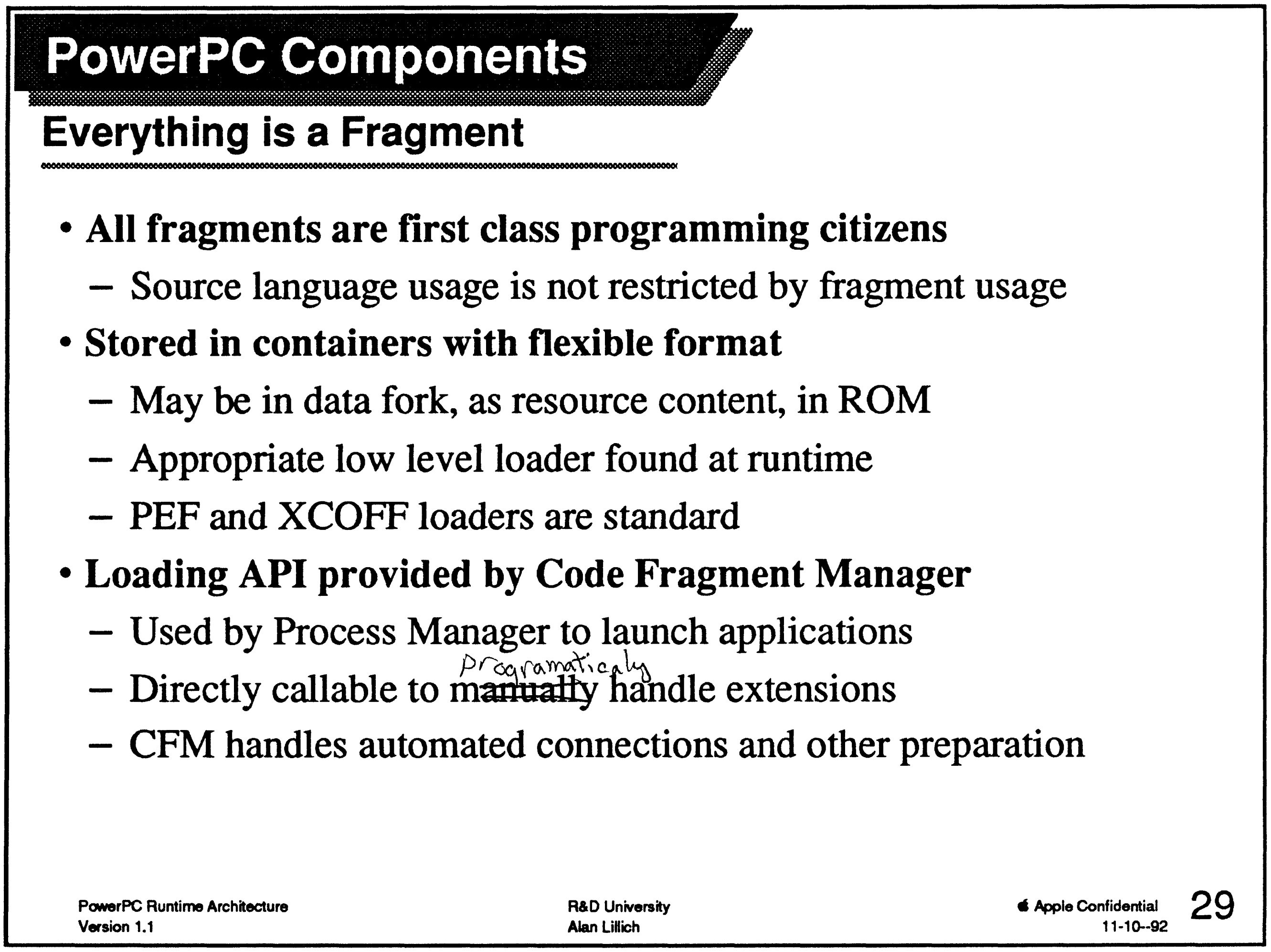
Everything is a Fragment



PowerPC Components

Fragment Usage Differences

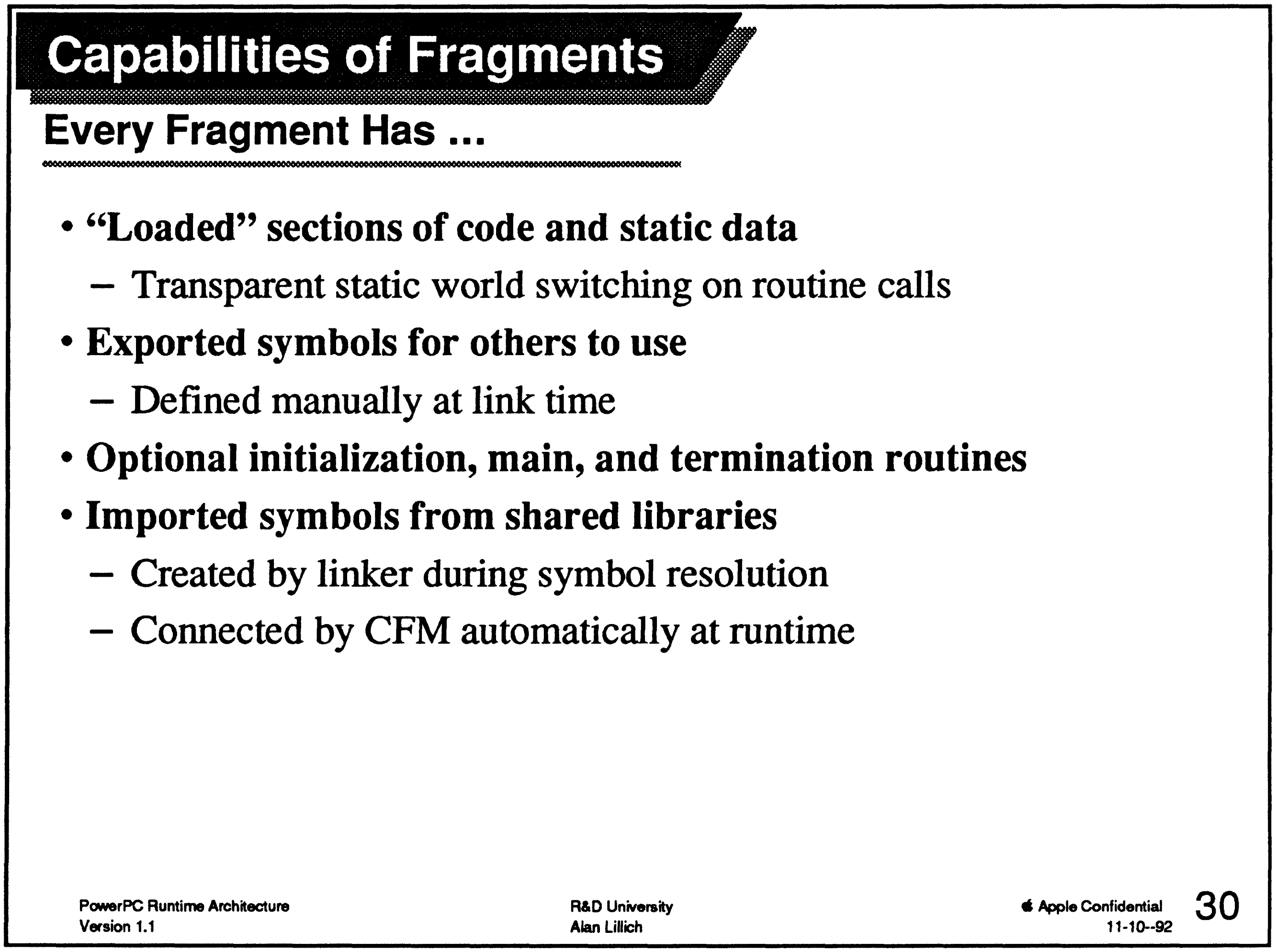
- **Application**
  - Extra information in “SIZE” resource, etc.
  - Stored in data fork of application file
- **Shared library**
  - Connected automatically at runtime to clients
  - Often stored in data fork of library file or ROM
- **Extension**
  - Connected by explicit request
  - Often stored in a resource



PowerPC Components

Everything is a Fragment

- **All fragments are first class programming citizens**
  - Source language usage is not restricted by fragment usage
- **Stored in containers with flexible format**
  - May be in data fork, as resource content, in ROM
  - Appropriate low level loader found at runtime
  - PEF and XCOFF loaders are standard
- **Loading API provided by Code Fragment Manager**
  - Used by Process Manager to launch applications
  - Directly callable to ~~manually~~ programaticaly handle extensions
  - CFM handles automated connections and other preparation



Capabilities of Fragments

Every Fragment Has ...

- **“Loaded” sections of code and static data**
  - Transparent static world switching on routine calls
- **Exported symbols for others to use**
  - Defined manually at link time
- **Optional initialization, main, and termination routines**
- **Imported symbols from shared libraries**
  - Created by linker during symbol resolution
  - Connected by CFM automatically at runtime

Capabilities of Fragments

Loaded Sections

- **Code**
  - Pure code ⇒ ROMable & directly pageable
  - Position independent
    - Code can be placed anywhere ⇒ no absolute branches in code
    - Data can be placed anywhere ⇒ no absolute data addresses in code
- **Static Data**
  - Contains pointers for position independence of code
  - Flexible sharing of data sections:
    - Global: One system-wide copy
    - Context: One copy per context (context ≈ application; the norm)
    - Never: Fresh copy for each load request

Fragment Reality Today

... and in the Immediate Future

- **Conventional languages need only two sections**
- **At most one section of code per fragment**
  - Not intended for 680x0–style segmentation
  - Data–only fragments are useful
- **One section of static data**
  - Includes both ".data" and ".bss"
  - Code–only fragments are not "conventional"

Exports and Imports

Both are created by the linker

- **Exports**
  - Definition: a symbol provided to the outside world by a fragment. It has a name and location (section + offset.) Used by imports. May be looked-up manually.
  - You give linker a list of global symbols to be exported
  - Both routines and static data can be exported
  - (FYI: Actually only data symbols are exported)

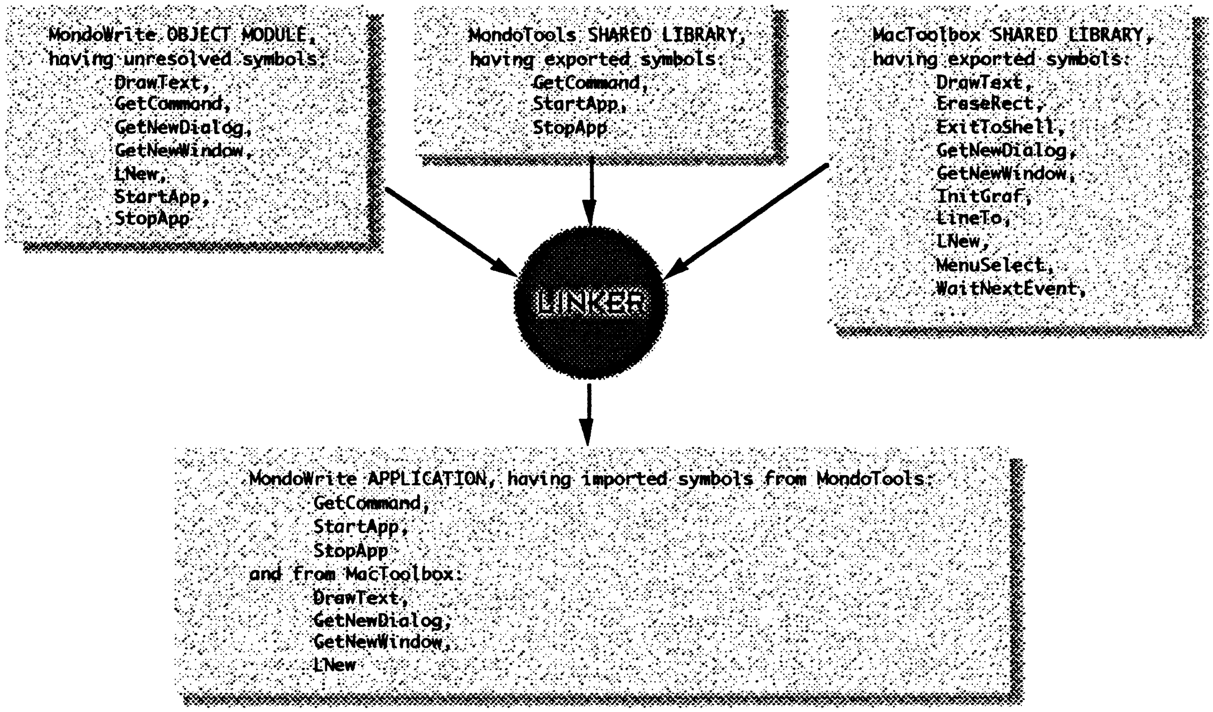
Exports and Imports

Both are created by the linker

- **Imports**
  - Definition: a reference in a fragment to an export from a shared library. Created at link time as part of the esternal symbol resolution. Final address binding is automatically perfrom at runtime.
  - You give linker shared libraries to link with
  - Shared library exports used for global symbol resolution
  - Symbols resolved to exports become imports
  - Imports recorded as library–name/export–name pair
  - Linker resolves symbols, CFM binds addresses

Exports and Imports

Linker Creation of Imports

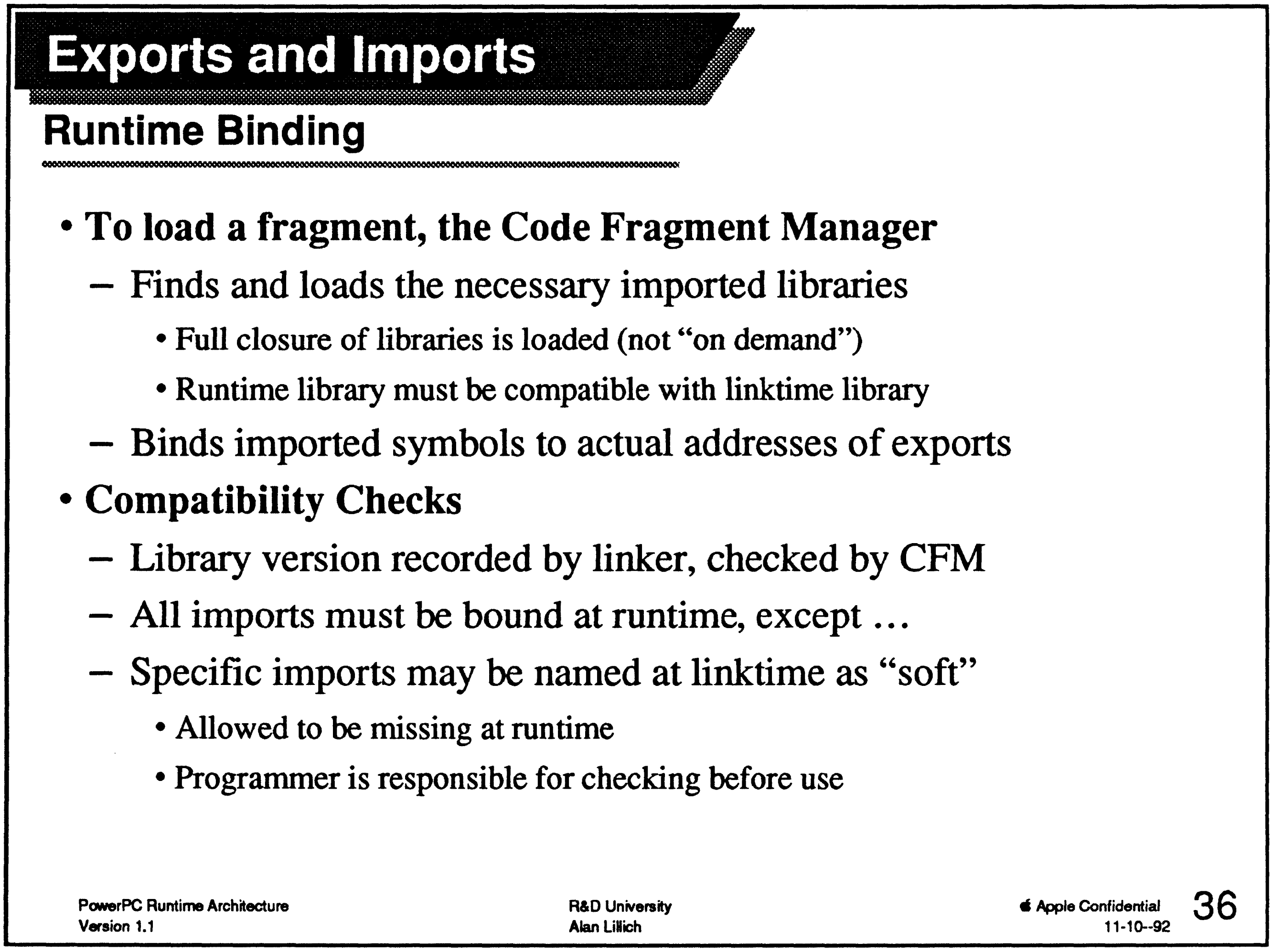


```
MondoWrite OBJECT MODULE,
having unresolved symbols:
        DrawText,
        GetCommand,
        GetNewDialog,
        GetNewWindow,
        LNew,
        StartApp,
        StopApp
```

```
MondoTools SHARED LIBRARY,
having exported symbols:
        GetCommand,
        StartApp,
        StopApp
```

```
MacToolbox SHARED LIBRARY,
having exported symbols:
        DrawText,
        EraseRect,
        ExitToShell,
        GetNewDialog,
        GetNewWindow,
        InitGraf,
        LineTo,
        LNew,
        MenuSelect,
        WaitNextEvent,
        . . .
```

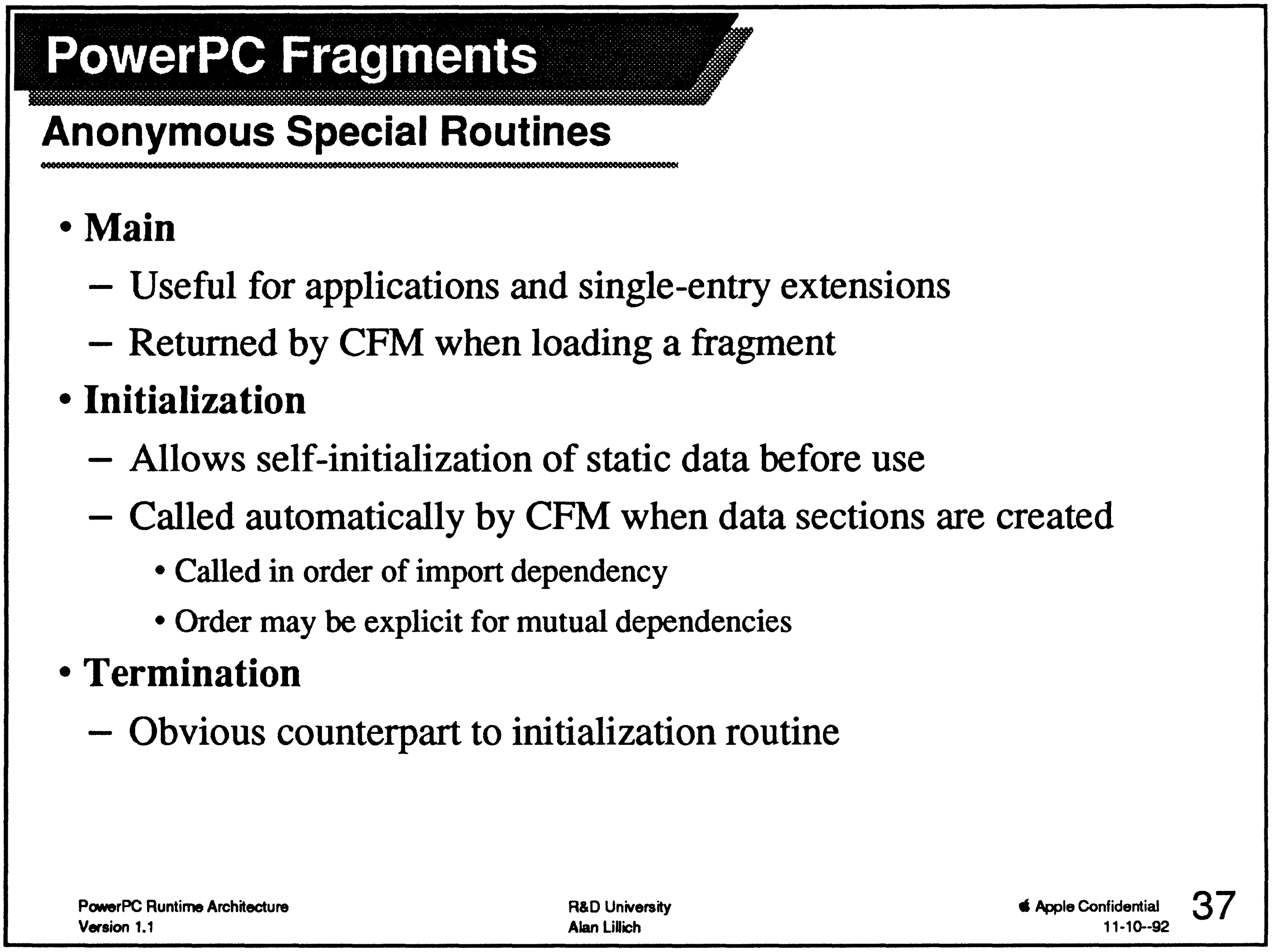
```
MondoWrite APPLICATION, having imported symbols from MondoTools:
        GetCommand,
        StartApp,
        StopApp
and from MacToolbox:
        DrawText,
        GetNewDialog,
        GetNewWindow,
        LNew
```



Exports and Imports

Runtime Binding

- **To load a fragment, the Code Fragment Manager**
  - Finds and loads the necessary imported libraries
    - Full closure of libraries is loaded (not “on demand”)
    - Runtime library must be compatible with linktime library
  - Binds imported symbols to actual addresses of exports
- **Compatibility Checks**
  - Library version recorded by linker, checked by CFM
  - All imports must be bound at runtime, except ...
  - Specific imports may be named at linktime as “soft”
    - Allowed to be missing at runtime
    - Programmer is responsible for checking before use



PowerPC Fragments

Anonymous Special Routines

- **Main**
  - Useful for applications and single-entry extensions
  - Returned by CFM when loading a fragment
- **Initialization**
  - Allows self-initialization of static data before use
  - Called automatically by CFM when data sections are created
    - Called in order of import dependency
    - Order may be explicit for mutual dependencies
- **Termination**
  - Obvious counterpart to initialization routine

Fragments in Action

Building a Shared Library

```
// MondoTools.h:

void StartApp ();
void StopApp ();

typedef struct {short kind;
                /* ... */ } CommandType, *CommandTypePtr;
void GetCommand (CommandTypePtr theCommand, int timeOut);
```

Fragments in Action

Building a Shared Library

```
// MondoTools.c

#include <MacToolbox.h>
#include "MondoTools.h"

void StartApp ()
{
        ...
}

void StopApp ()
{
        ...
}

void GetCommand (CommandTypePtr theCommand, int timeOut)
{
        ...
}
```

Fragments in Action

Building a Shared Library

```
// MondoTools.export
StartApp
StopApp
GetCommand
```

```
// MondoTools.build.IBM
cc -c MondoTools.c
ld MondoTools.o MacToolbox.XCOFF \
    -o MondoTools.XCOFF -bM:SRE -bE:MondoTools.export
cvtx2p MondoTools.XCOFF -o MondoTools
```

Fragments in Action

Building an Application

```
// MondoWrite.c

#define Boolean int
#define false 0
#define true 1
#define kNullCommand 0
#define kExitCommand 1

#include <MacToolbox.h>
#include "MondoTools.h"

static void DoTextCommand (CommandTypePtr theCommand)
{
        ...
        DrawText (...);
        ...
}

static void DoListCommand (CommandTypePtr theCommand)
{
        ...
        LNew (...);
        ...
}
```

Fragments in Action

Building an Application

```
void Main ()
{
        Boolean done = false
        CommandType thisCommand;

        StartApp ();

        while(!done) {
                GetCommand(&thisCommand,1);
                switch (thisCommand.kind) {
                          case kNullCommand.break;
                          case kTextCommand: DoTextCommand (...); break;
                          case kListCommand: DoListCommand (...); break;
                          case kExitCommand: done = true; break;
                };
        };

        StopApp ();
}
```

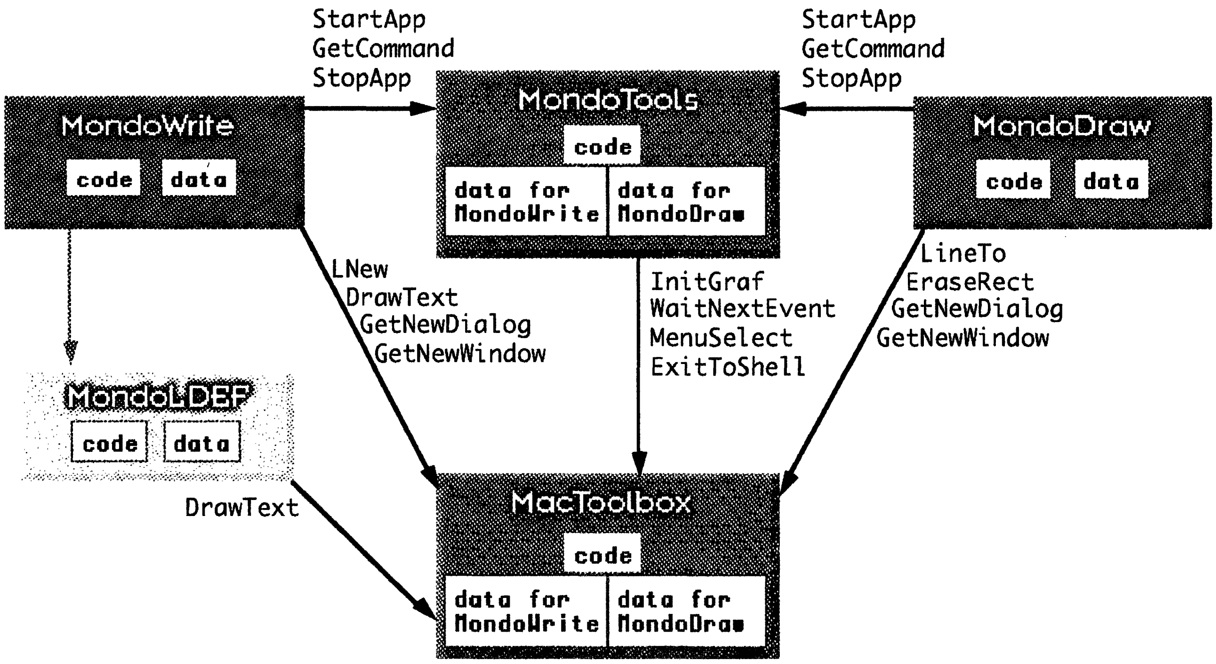
Fragments in Action

Building an Application

```
// MondoWrite.build.IBM
cc -c MondoWrite.c
ld MondoWrite.o MacToolbox.XCOFF MondoTools.XCOFF \
    -o MondoWrite.XCOFF -eMain
cvtx2p MondoWrite.XCOFF -o MondoWrite
```

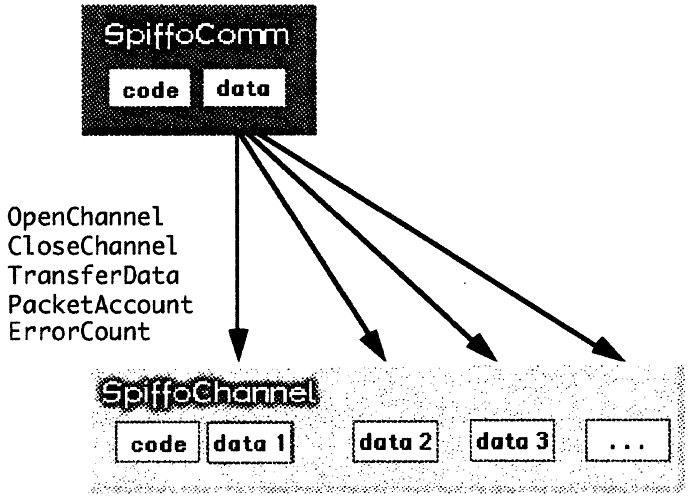
Fragments in Action

App, Shared Libs, & Extension



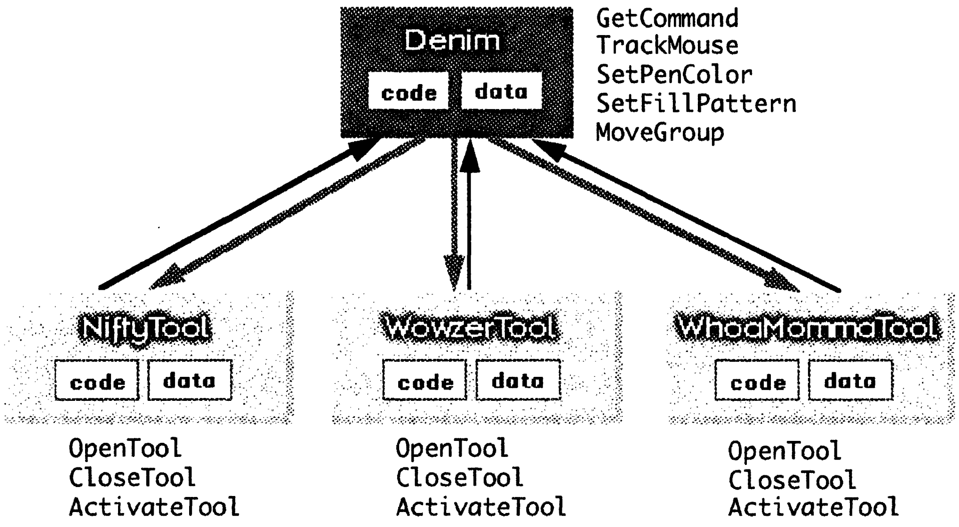
Fragments in Action

Extension Exporting Full API



Fragments in Action

App with Custom Extensions



GetCommand
TrackMouse
SetPenColor
SetFillPattern
MoveGroup

OpenTool
CloseTool
ActivateTool

OpenTool
CloseTool
ActivateTool

OpenTool
CloseTool
ActivateTool

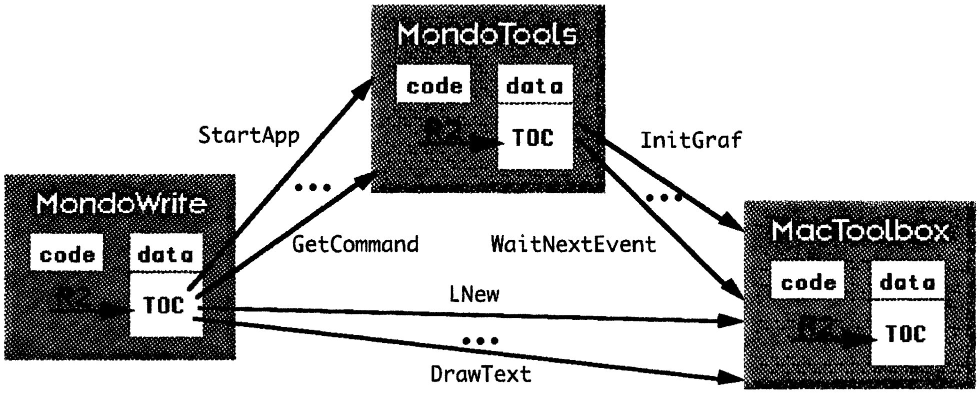
PowerPC Fragments

- **Questions?**

Global Addressing

The Table of Contents (TOC)

- **The TOC is ...**
  - The code's gateway to the world
    - More akin to personal address book than table of contents
  - Buried in the static data
    - Addressed through a dedicated register (R2 a.k.a. RTOC)

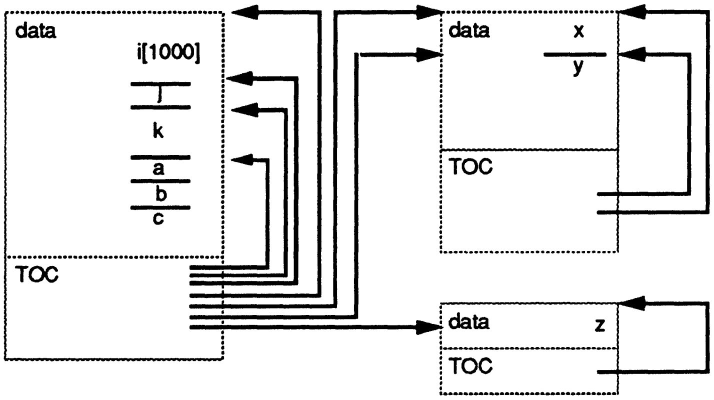


Global Addressing

The Table of Contents (TOC)

- **The TOC collects pointers to support addressing model**
  - Localizes memory modified during loading
  - Static data is addressed indirectly through the TOC
  - Imported routines are called indirectly through the TOC
    - (We'll see later that they actually use pointers to data to do this)

```
int i[1000], j, k;
static int a, b, c;
extern int x, y, z;
```

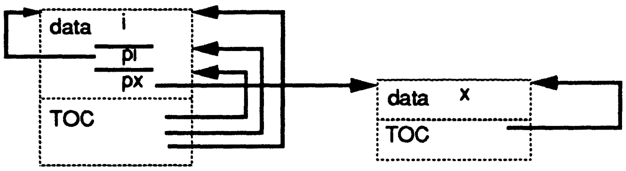


Global Addressing

The Table of Contents (TOC)

- **Explicit (source) pointers do not use the TOC**
  - Initialized static pointers get relocated (e.g. C++ VTables)

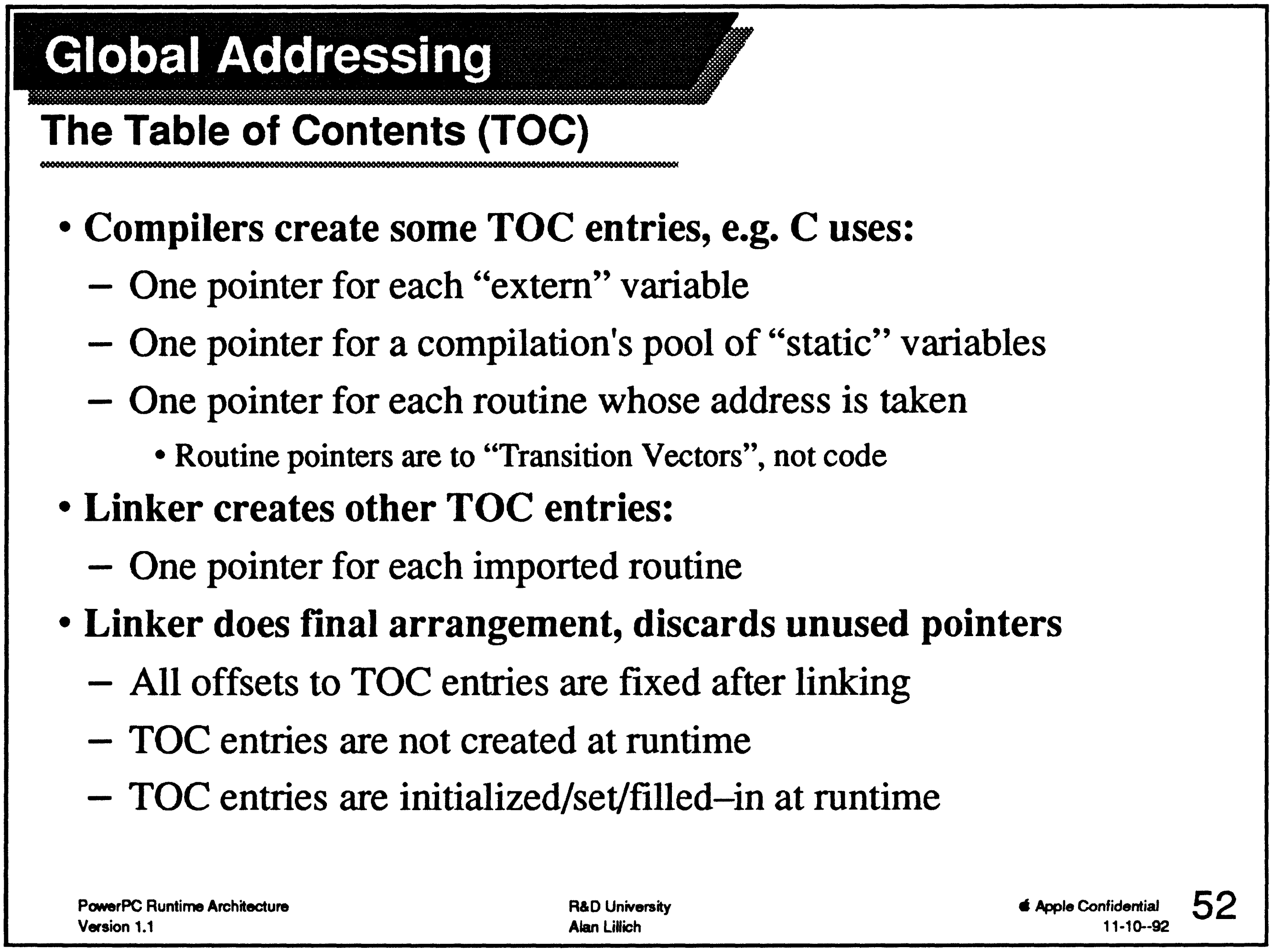
```
int i=0;
extern int x;

int *pi = &i;
int *px = &x;
```



Global Addressing

The Table of Contents (TOC)

- **The TOC supports ...**
  - Purity and position independence of code
  - Separate static worlds ...
    - For separate fragments
    - For separate instances of one fragment
  - Addressing of large numbers of static data items
  - Unbounded size for individual static data items
- **The TOC really belongs to routines, not fragments**
  - It is where RTOC points when a routine is called
  - RTOC is switched as part of cross-TOC calls
  - A linker is free to build multiple TOCs in one fragment



Global Addressing

The Table of Contents (TOC)

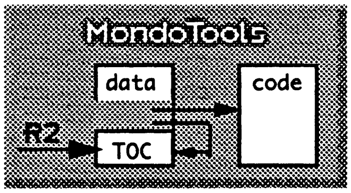
- **Compilers create some TOC entries, e.g. C uses:**
  - One pointer for each “extern” variable
  - One pointer for a compilation's pool of “static” variables
  - One pointer for each routine whose address is taken
    - Routine pointers are to “Transition Vectors”, not code
- **Linker creates other TOC entries:**
  - One pointer for each imported routine
- **Linker does final arrangement, discards unused pointers**
  - All offsets to TOC entries are fixed after linking
  - TOC entries are not created at runtime
  - TOC entries are initialized/set/filled–in at runtime

Global Addressing

Transition Vector (TVector)

- Two or three word structure:
  - Entry point
  - TOC address
  - [Environment word]

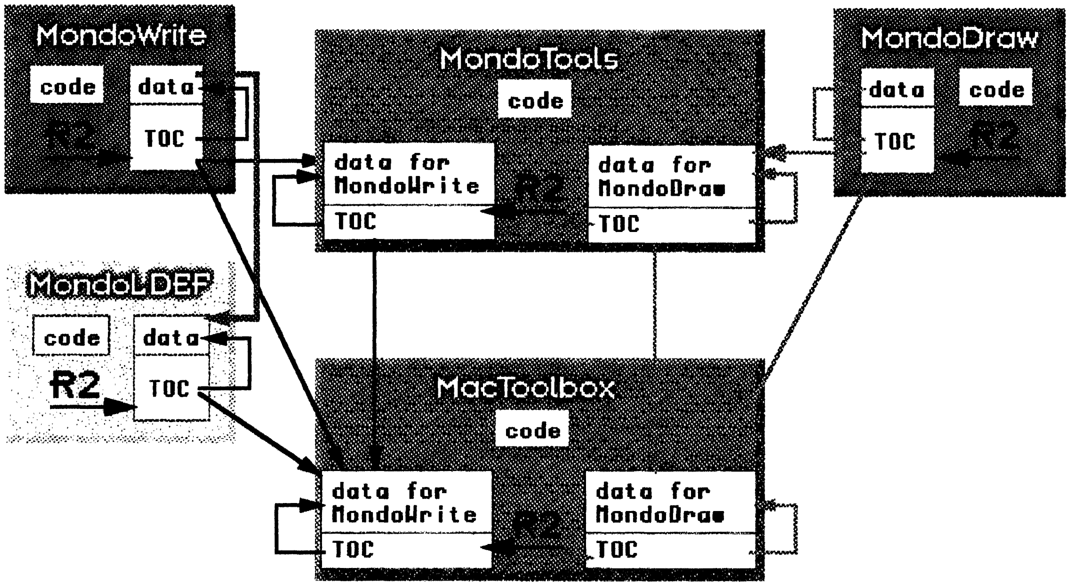
```
typedef struct {
    Address Entry_Point;
    Address TOC_Pointer;
    Address Environment;
} Transition_Vector;
```



- Created by compilers
- In static data, same fragment as code
- The address of a routine is the address of the TVector
  - Export TVector address, not code address
- Not the "Routine Descriptor" of Mixed Mode

Fragments in Action

App & Friends, with TOC



Global Addressing

- **Questions?**

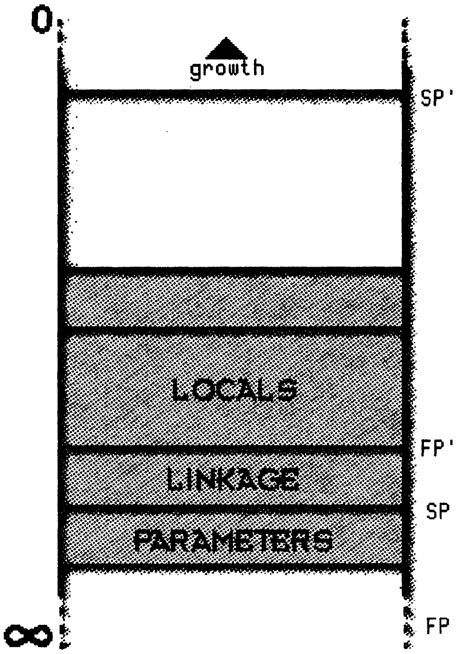
Calling Conventions

- **Stack Usage**
  - Radically different from 680x0 approach
- **Parameter Passing**
  - Largely register–based, influenced by stack model
- **Code Generation for Calls**
  - Compiler & Linker cooperate on cross–TOC calls

Stack Usage

680x0 Stack Model

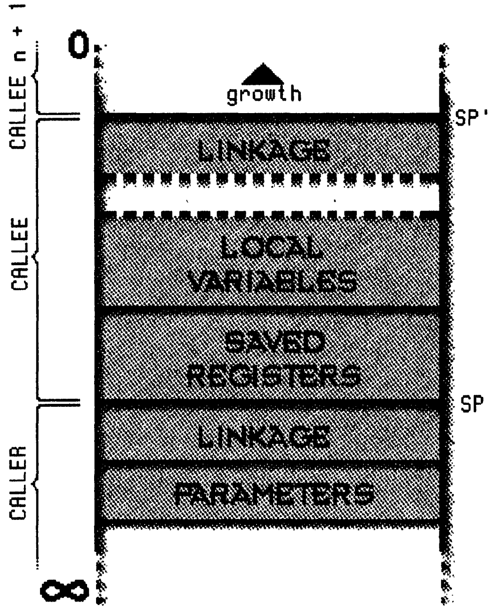
- **Conventional grow–down stack**
  - Separate frame pointer (FP) and stack pointer (SP)
  - Parameters above FP, locals below FP
  - Push and pop at will using SP
- **Pascal and C have different protocols (Unnecessarily!)**
  - Order of parameter processing
  - Which side pops parameters
  - Where function results are returned
- **Hardware enforces 16–bit alignment**
  - 32–bit alignment preferred



Stack Usage

PowerPC Stack Model

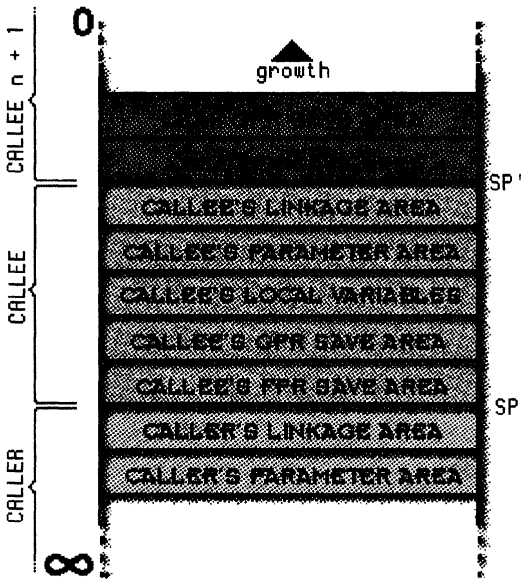
- **Still a grow–down stack, but ...**
  - – Single top of stack pointer (SP)
  - – Special areas for linkage, parameters, locals, etc.
  - – No “trivial” pushing and popping
- **Uniform usage for all languages (We hope!)**
- **Alignment enforced by software**
- **Organized to reduce common–case memory references**



PowerPC Stack Frames

Contents of Special Areas

- The “RedZone”™
- Callee's linkage area
- Callee's parameter area
- Callee's local variables
- Callee's FPR/GPR save areas
- Caller's linkage area
- Caller's parameter area



0

growth

CALLEE n + 1

SP'

CALLEE'S LINKAGE AREA

CALLEE'S PARAMETER AREA

CALLEE'S LOCAL VARIABLES

CALLEE'S GPR SAVE AREA

CALLEE'S FPR SAVE AREA

CALLEE

SP

CALLER'S LINKAGE AREA

CALLER'S PARAMETER AREA

CALLER

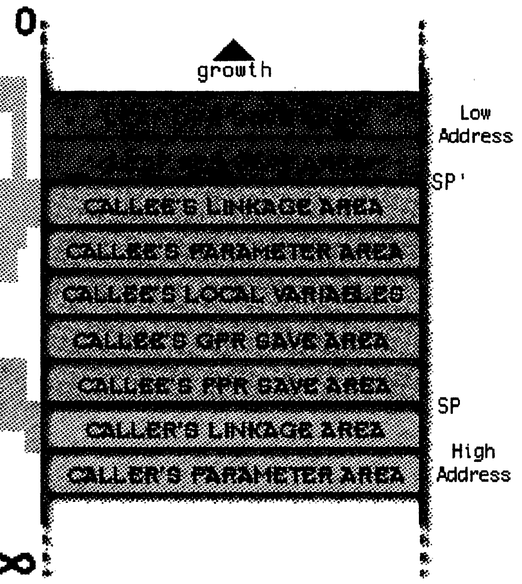
∞

PowerPC Stack Frames

Stack Linkage Area

Set in own area at frame creation	0 Saved SP
	4 Saved CR
	8 Saved LR
Reserved (not used at present)	12 reserved
Used by “patching” mechanism	16 reserved
Set in own area by cross-TOC glue	20 Saved RTOC

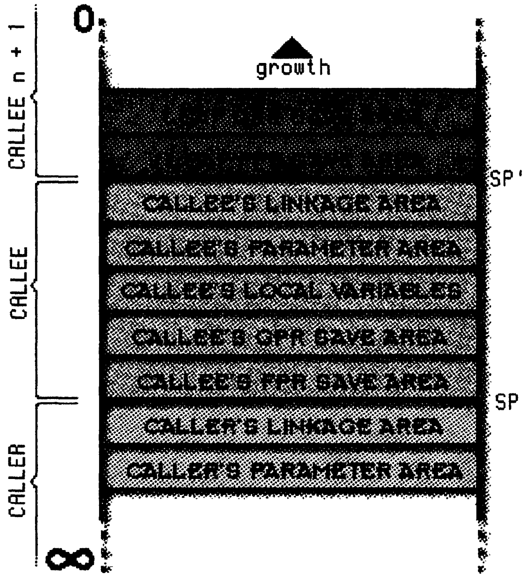
	0 Saved SP
Set in caller's area by callee's prolog	4 Saved CR
Set in caller's area by callee's prolog	8 Saved LR
	12 reserved
	16 reserved
	20 Saved RTOC



PowerPC Stack Frames

General Responsibilities

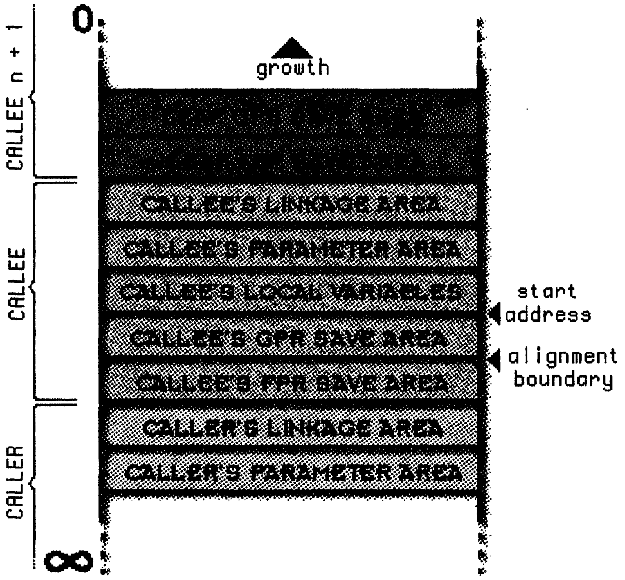
- Caller prepares parameters then performs call
- RTOC saved in caller's linkage area (by glue or caller)
- Callee saves link and condition registers in caller's linkage area
- Callee saves nonvolatile FPRs and GPRs on the stack
- Callee allocates frame, preserving alignment and linkage (old SP)
- Order of callee actions is by convention, not requirement



PowerPC Stack Frames

Alignment Issues

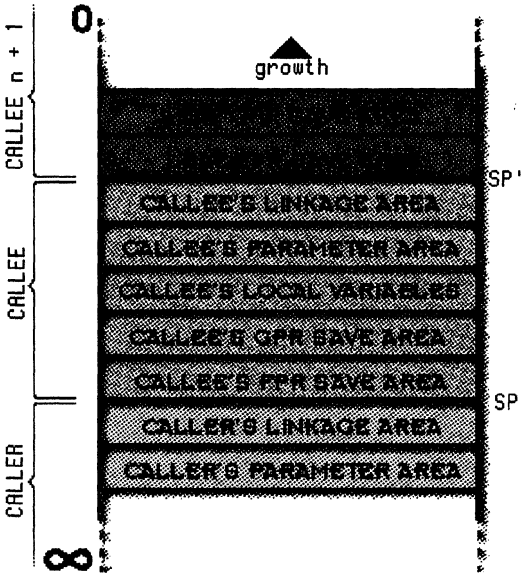
- **Alignment is maintained totally by software**
- **GPR Load/Store multiple have hardware impact**
  - Require “natural” alignment
  - Prefer quadword (16-byte) alignment at high address end
  - Instruction:
    `stmw start_reg, start_address`
- **SP presumed to be kept quadword aligned**



PowerPC Stack Model

Register Saving & Restoring

- **Callee saves & restores almost all nonvolatile registers**
  - Link (LR) & condition (CR) registers in caller's linkage area
  - "High" GPRs (13–31) & FPRs (14–31) in own save areas
- **RTOC is special**
  - Saved "between" caller and callee in caller's linkage area
  - Restored by caller immediately upon return
- **Leaf routines use "top" linkage area and the RedZone™**



PowerPC Stack Model

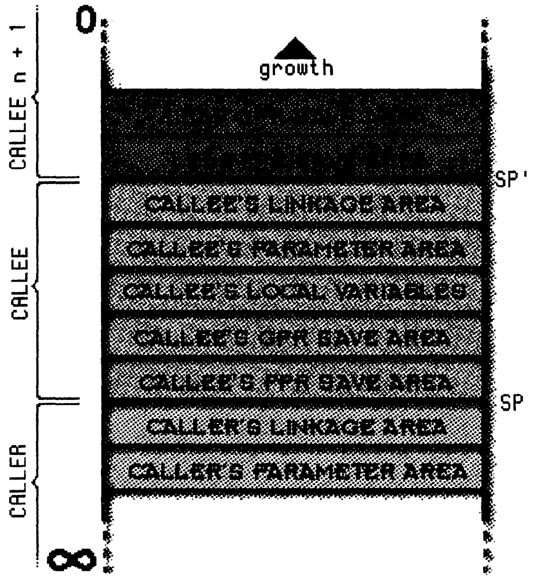
Interrupt Issues

- **The RedZone™**
  - Must allow for maximum use
  - Decrement SP by 224 before using stack
    - 19x4 (GPRs) + 18x8 (FPRs) rounded to quadword
- **Alignment**
  - Assume preserved by PowerPC code
  - Assume not preserved by emulated code
  - Save current value and clear low 4 bits of SP
    - Don't save old SP on stack without skipping RedZone™ first!

Calling Conventions

Parameter Passing

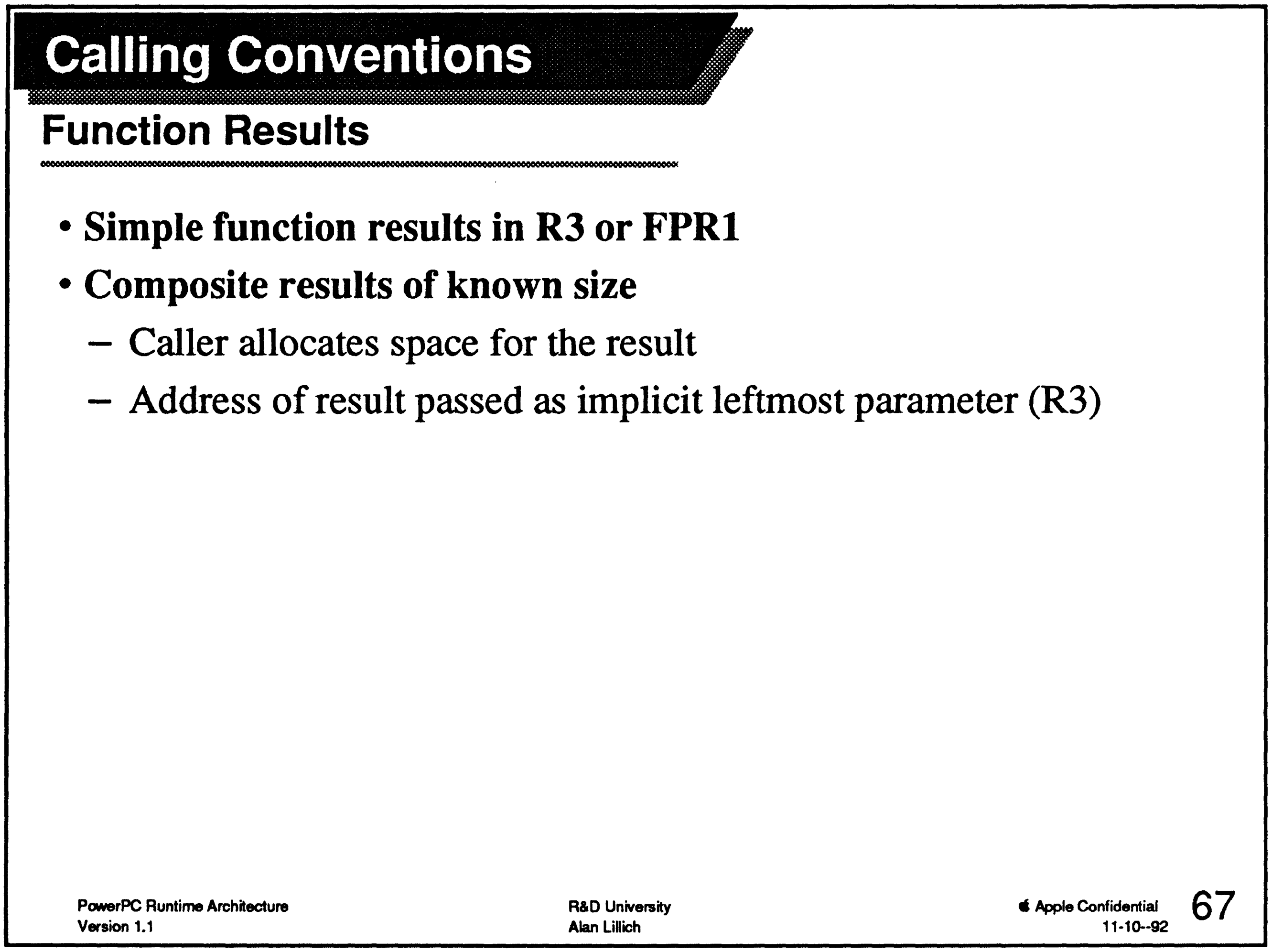
- **Single parameter area in each frame**
  - Used by caller to prepare parameters
  - Used by callee to access parameters
  - Large enough for largest parameter list



Calling Conventions

Parameter Passing

- **Layout parameters like a record**
  - Leftmost parameter as first "field"
  - Each field aligned to a word boundary
  - Small "integers" extended to a word
  - Composites not affected internally
- **Some parameter values passed in registers**
  - First 8 words mapped to GPRs 3–10, except …
  - First 13 "visible" floats passed in FPR 1–13



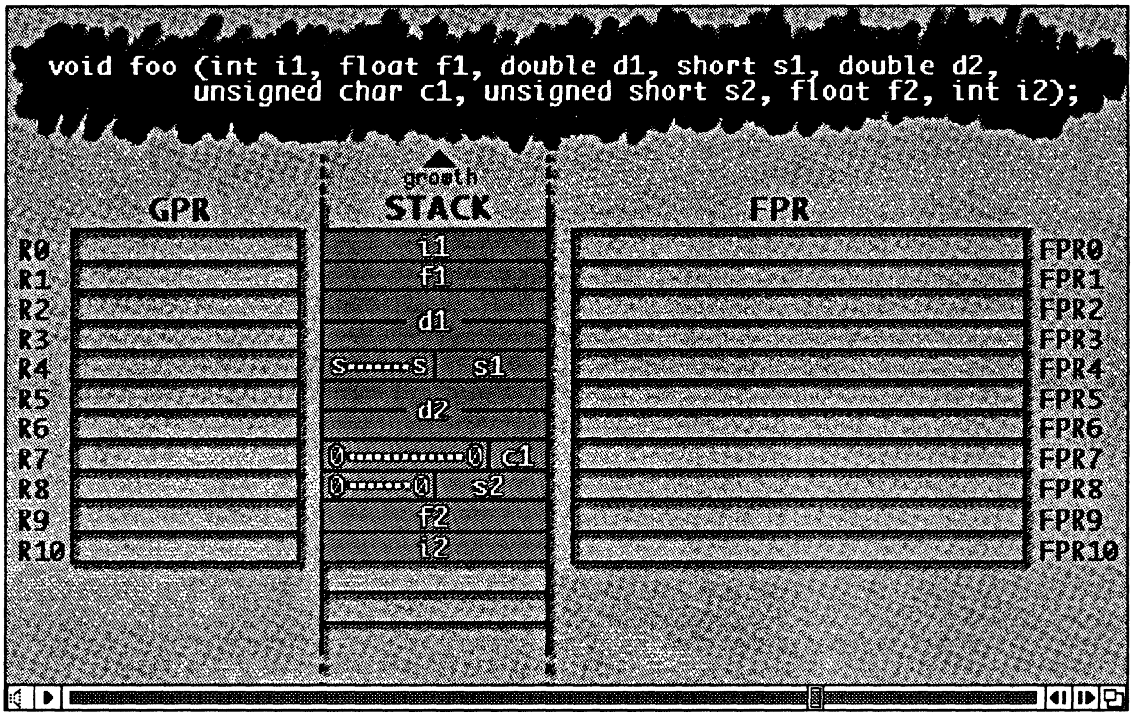
Calling Conventions

Function Results

- **Simple function results in R3 or FPR1**
- **Composite results of known size**
  - Caller allocates space for the result
  - Address of result passed as implicit leftmost parameter (R3)

Calling Conventions

GPR and FPR examples



```
void foo (int i1, float f1, double d1, short s1, double d2,
          unsigned char c1, unsigned short s2, float f2, int i2);
```

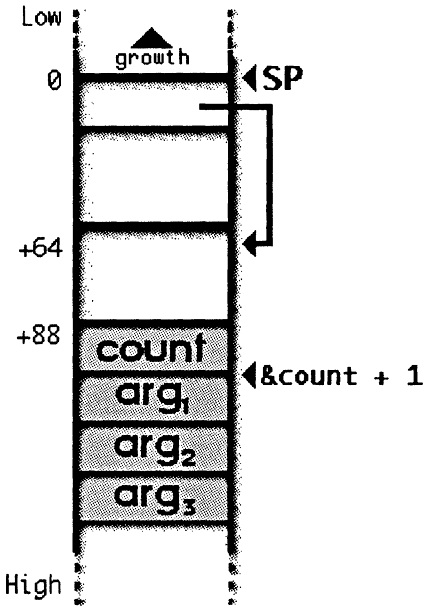
Calling Conventions

Parameter Hacks for C

- **Variable numbers of parameters ⇒ Minimum of 8 word parameter area**
  - Callee doesn't know how many were passed
  - Callee saves R3 through R10 into the parameter area
  - Callee walks through parameter area to access values

```
int Sum (int count, ...) {
    int result = 0;
    int *arg = &count + 1;
    while (--count >= 0) {
        result += *arg;
        arg++;
    }
    return result;
}

void main () {
    int total;
    total = Sum (3, 2, 4, 6);
}
```

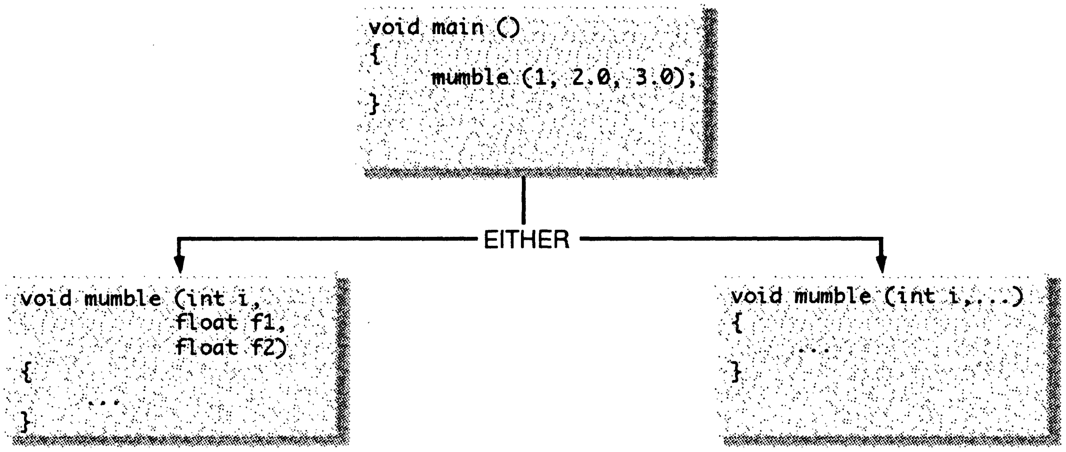
```
.Sum:  stwu   SP,-64(SP)
       stw    R3,88(SP)
       stw    R4,92(SP)
       stw    R5,96(SP)
       stw    R6,100(SP)
       stw    R7,104(SP)
       stw    R8,108(SP)
       stw    R9,112(SP)
       stw    R10,116(SP)
       addi   R3,0(R0)
       stw    R3,56(SP)
       addi   R3,88(SP)
       addic  R3,R3,4
       stw    R3,60(SP)
       ...
```



Calling Conventions

Parameter Hacks for C

- **Lack of prototypes ⇒ Floats in both GPR and FPR**
  - Caller doesn't know if callee expects floats or not
  - Callee always knows what it expects



```
void main ()
{
    mumble (1, 2.0, 3.0);
}
```

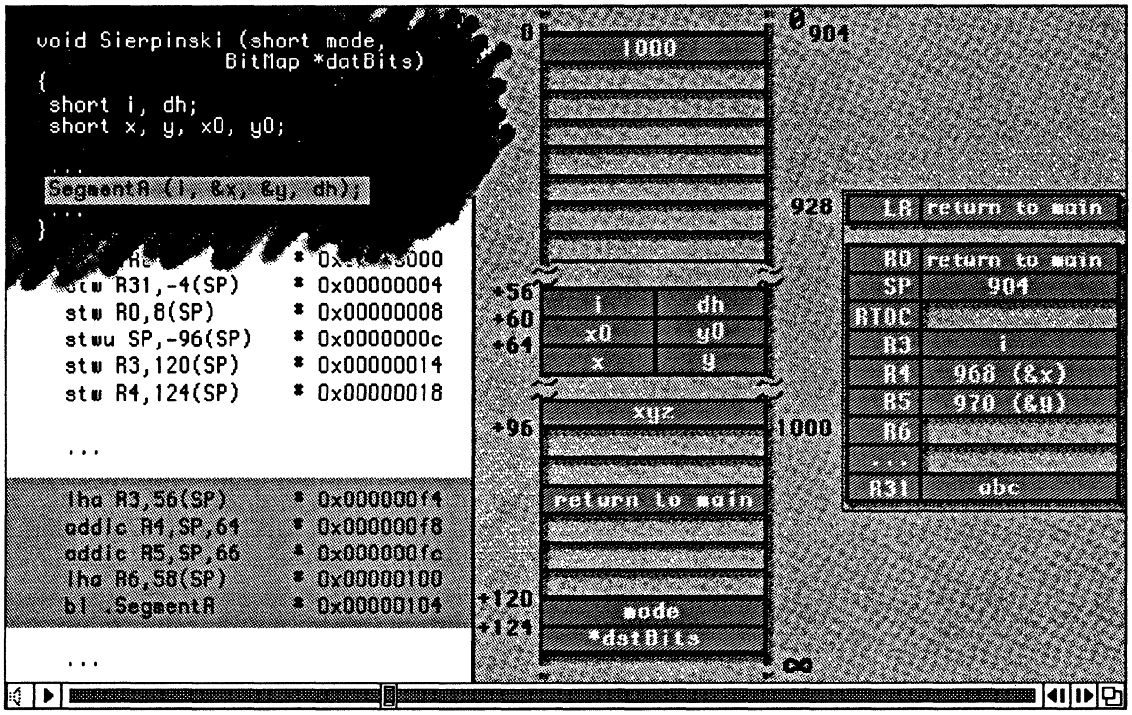
EITHER

```
void mumble (int i,
             float f1,
             float f2)
{
   ...
}
```

```
void mumble (int i,...)
{
   ...
}
```

Parameter Passing

Example Showing Code & Stack



```
void Sierpinski (short mode,
                BitMap *datBits)
{
 short i, dh;
 short x, y, x0, y0;

 ...
 SegmentA (i, &x, &y, dh);
 ...
}
   [illegible] R0            * 0x[illegible]000
   stw R31,-4(SP)        * 0x00000004
   stw R0,8(SP)          * 0x00000008
   stwu SP,-96(SP)       * 0x0000000c
   stw R3,120(SP)        * 0x00000014
   stw R4,124(SP)        * 0x00000018

   ...

   lha R3,56(SP)         * 0x000000f4
   addic R4,SP,64        * 0x000000f8
   addic R5,SP,66        * 0x000000fc
   lha R6,58(SP)         * 0x00000100
   bl .SegmentA          * 0x00000104

   ...
```

LR	return to main
R0	return to main
SP	904
RTOC	
R3	i
R4	968 (&x)
R5	970 (&y)
R6	
...	
R31	abc

Calling Conventions

Code Generation for Calls

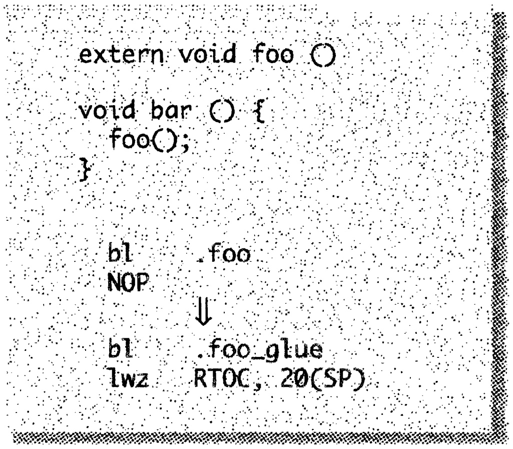
- **Local call, a "compiled–in" call without a TOC switch**
  - Direct branch to callee

```
void foo () {
  return;
}

void bar () {
  foo();
}


    bl    .foo
```

Calling Conventions

Code Generation for Calls

- **Non–local call, “compiled–in” call with a TOC switch**
  - Compiler prepares, linker finishes cross–TOC connection



```
extern void foo ()

void bar () {
  foo();
}


  bl     .foo
  NOP
          ⇓
  bl     .foo_glue
  lwz    RTOC, 20(SP)
```

Calling Conventions

Code Generation for Calls

- **Pointer–based call, a call through a routine pointer**
  - Very similar to cross–TOC case

```
void (*foo) ();

void bar () {
  (*foo) ();
}


  lwz   R11, foo #TVector address
  bl    ptr_glue
  lwz   RTOC, 20(SP)
```

Code Generation for Calls

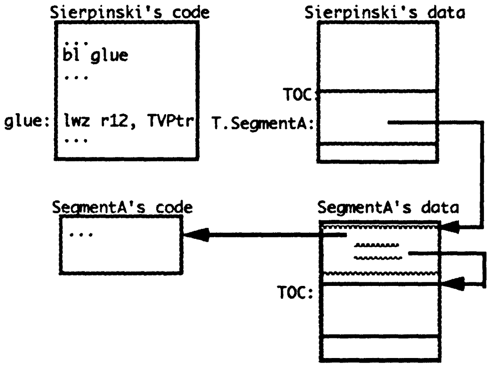
- **Cross-TOC Call Details**
- **Compiler generates RTOC reload slot w/ NOP**
- **Linker “fixes” RTOC reload, appends custom glue**
- **Linker may create TOC pointer to TVector**
- **Optional “environment” word ignored**

```
extern void SegmentA (...);

void Sierpinski (...)
{
   ...
   SegmentA(...);
   ...
}

.Sierpinski:
   ...
   bl      .SegmentA
   NOP     <<cror or ori>>
   ...
```

```
.Sierpinski:
   ...

   bl      .SegmentA_glue
   lwz     RTOC, 20(SP)

   ...

SegmentA_glue:
   lwz     R12, T.SegmentA(RTOC)
   stw     RTOC, 20(SP)
   lwz     R0, 0(R12)
   lwz     RTOC, 4(R12)
   mtctr   R0
   bctr
```

Code Generation for Calls

- Cross-TOC Call Details
- Compiler generates RTOC reload slot w/ NOP
- Linker "fixes" RTOC reload, appends custom glue
- Linker may create TOC pointer to TVector
- Optional "environment" word ignored



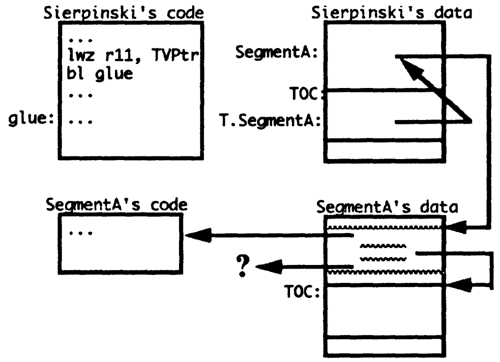
Code Generation for Calls

- Pointer–based Call Details
- Compiler calls standard glue, passing TVector address
- Environment word passed in R11

```
static void (*SegmentA) (...);

void Sierpinski (...)
{
    ...
    (*SegmentA)(...);
    ...
}

.Sierpinski:
    ...

    lwz     R3, T.SegmentA(RTOC)
    lwz     R11, 0(R3)
    bl      ._ptrglue
    lwz     RTOC, 20(SP)

    ...

._ptrglue:
    lwz     R0, 0(R11)
    stw     RTOC, 20(SP)
    mtctr   R0
    lwz     RTOC, 4(R11)
    lwz     R11, 8(R11)
    bctr
```

Code Generation for Calls

- Pointer–based Call Details
- Compiler calls standard glue, passing TVector address
- Environment word passed in R11



```
...
lwz r11, TVPtr
bl glue
...
glue: ...
```

Calling Conventions

- **Questions?**

Had Enough?

End of First Day

- **Tomorrow's agenda**
  - Object modules
  - IBM's "dis" tool
  - Sierpinski example
  - Shared libraries
  - System software issues
- **Questions?**

Day 2 Content

Object Modules

Sierpinski Code

System SW Issues

Shared Libraries

Schedule for Day 2

Second Day of Course

- Object modules
- IBM's “dis” tool
- Sierpinski example
- Shared libraries
- System software issues

Object Modules

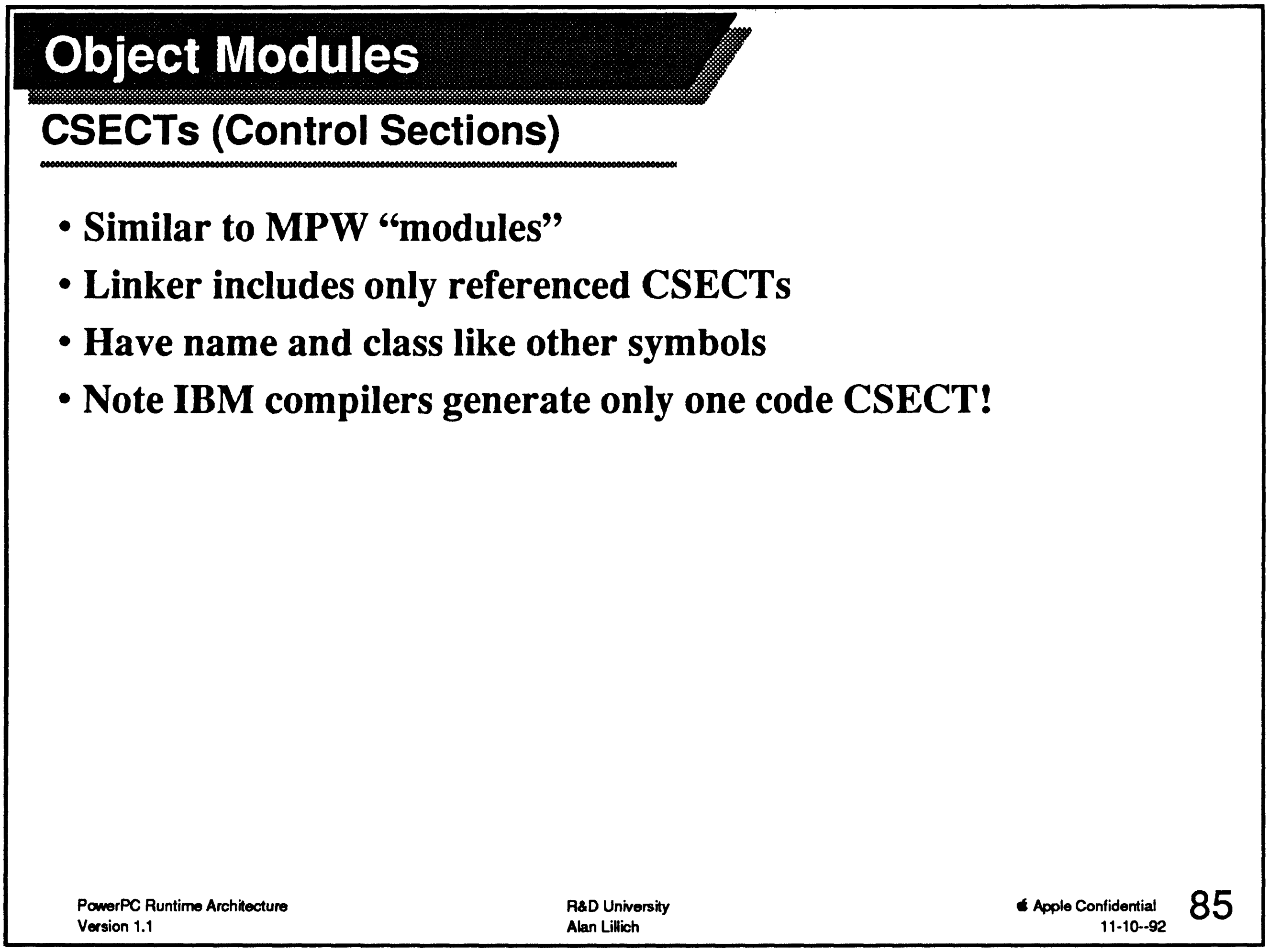
File Formats

- **Highlights, details later**
- **Compiler output is XCOFF**
- **IBM linker output is XCOFF**
- **Preferred Macintosh executable format is PEF**
  - Format defined by Apple to address XCOFF problems
  - Generate today via XCOFF⇒PEF conversion tool
  - Very similar in overall concept to XCOFF
  - Considerably smaller and faster to load
  - More expressive than XCOFF for runtime needs
  - Lacks defined object module capabilities

Object Modules

Symbol Names & Classes

- **Symbols are identified by both name and class**
  - Written as “name{class}” or “name[class]”
- **Common symbol classes:**
  - PR ⇒ code (program)
  - RW ⇒ initialized static data
  - BS ⇒ uninitialized static data
  - TC ⇒ TOC (note special TOC{TC0} symbol)
  - UA ⇒ unassigned (static data)
  - DS ⇒ transition vector (descriptor to IBM)
  - GL ⇒ linker created call glue



Object Modules

CSECTs (Control Sections)

- Similar to MPW “modules”
- Linker includes only referenced CSECTs
- Have name and class like other symbols
- Note IBM compilers generate only one code CSECT!

Object Modules

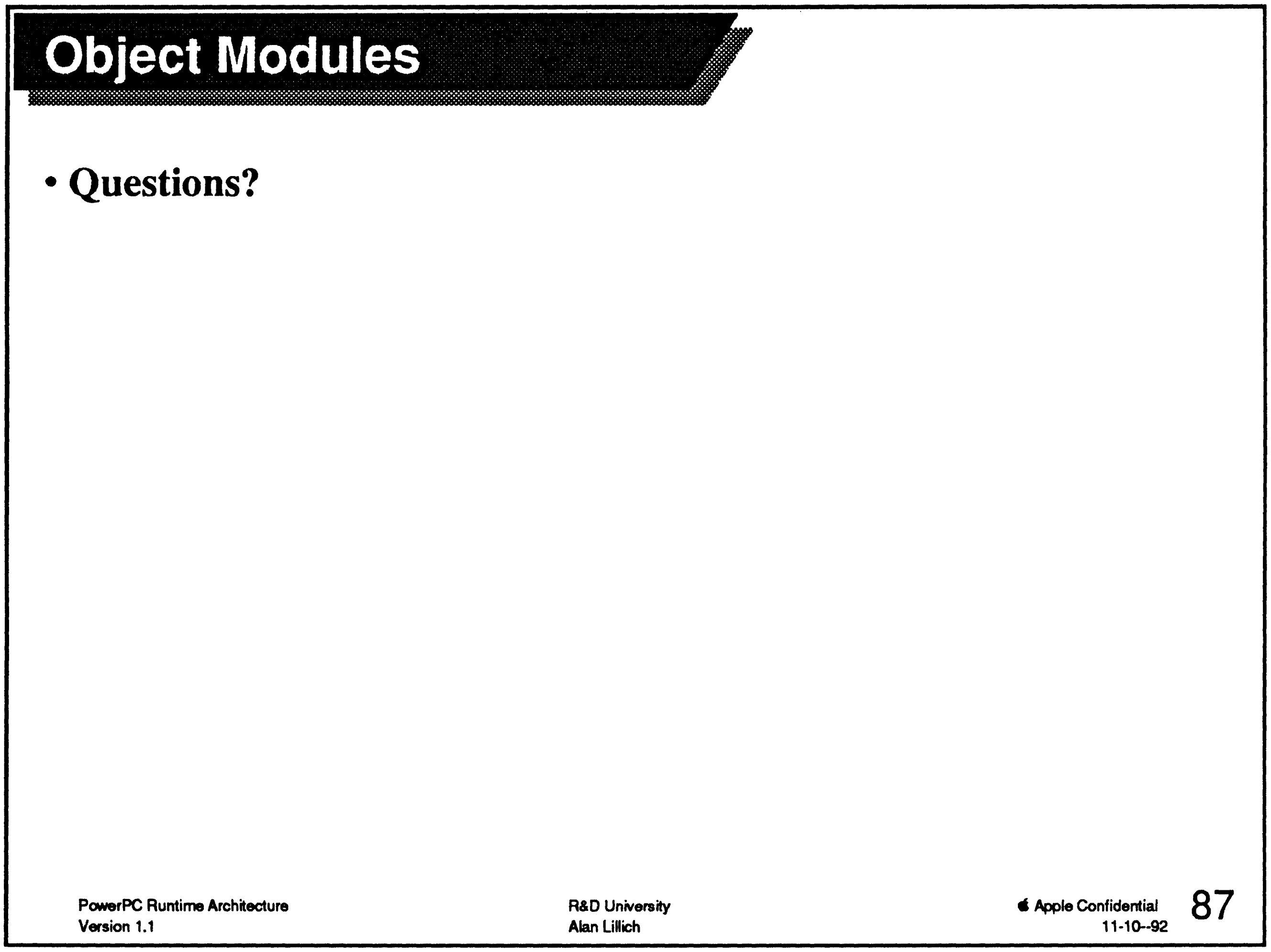
The “.tc” macro

- **Format:**

```
label   .tc     name, contents
```

- **Equivalent:**

```
        .csect  name{TC}
label   .long   contents
```



Object Modules

- **Questions?**

PowerPC Runtime Architecture
Version 1.1

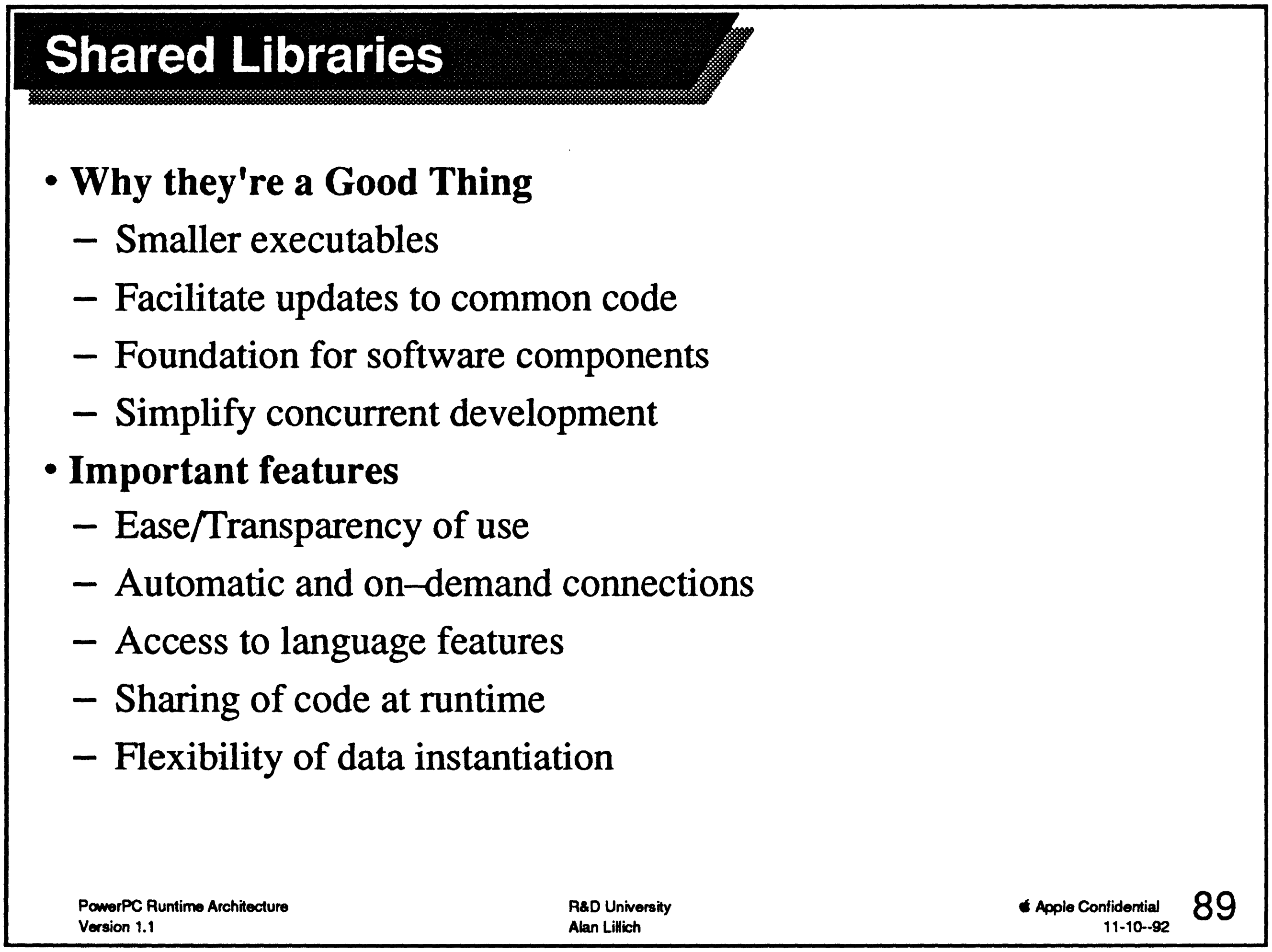
R&D University
Alan Lillich

 Apple Confidential 87
11-10--92

Review of Yesterday

Sierpinski Example

- (See handouts)

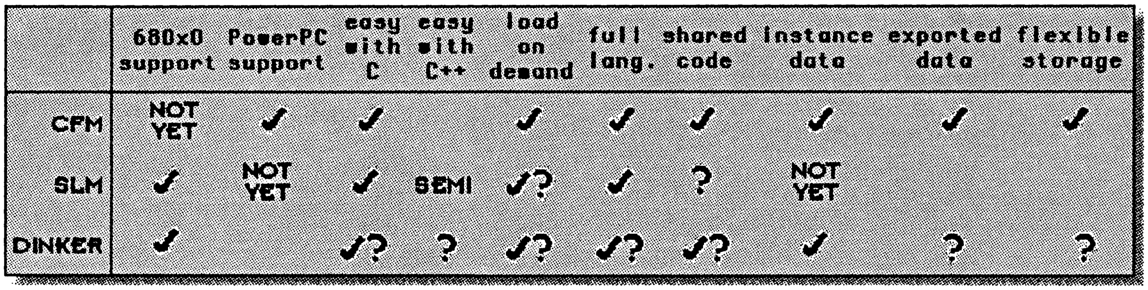


Shared Libraries

- **Why they're a Good Thing**
  - Smaller executables
  - Facilitate updates to common code
  - Foundation for software components
  - Simplify concurrent development
- **Important features**
  - Ease/Transparency of use
  - Automatic and on–demand connections
  - Access to language features
  - Sharing of code at runtime
  - Flexibility of data instantiation

Shared Libraries

- **Available Implementations**
  - CFM on PowerPC
  - SLM on 680x0
  - Dinker on 680x0
- **Feature Matrix**



	680x0 support	PowerPC support	easy with C	easy with C++	load on demand	full lang.	shared code	instance data	exported data	flexible storage
CFM	NOT YET	✓	✓		✓	✓	✓	✓	✓	✓
SLM	✓	NOT YET	✓	SEMI	✓?	✓	?	NOT YET		
DINKER	✓		✓?	?	✓?	✓?	✓?	✓	?	?

Shared Libraries

Status of CFM & CFL

- **Intrinsic part of PowerPC runtime**
- **In operation today**
- **Fully source transparent**
- **Most flexible data support**
- **Additional Features**
  - Bidirectional version checking
  - Supports alternate internal models
  - Integrated with Mixed Mode
  - Partial library updates

Shared Libraries

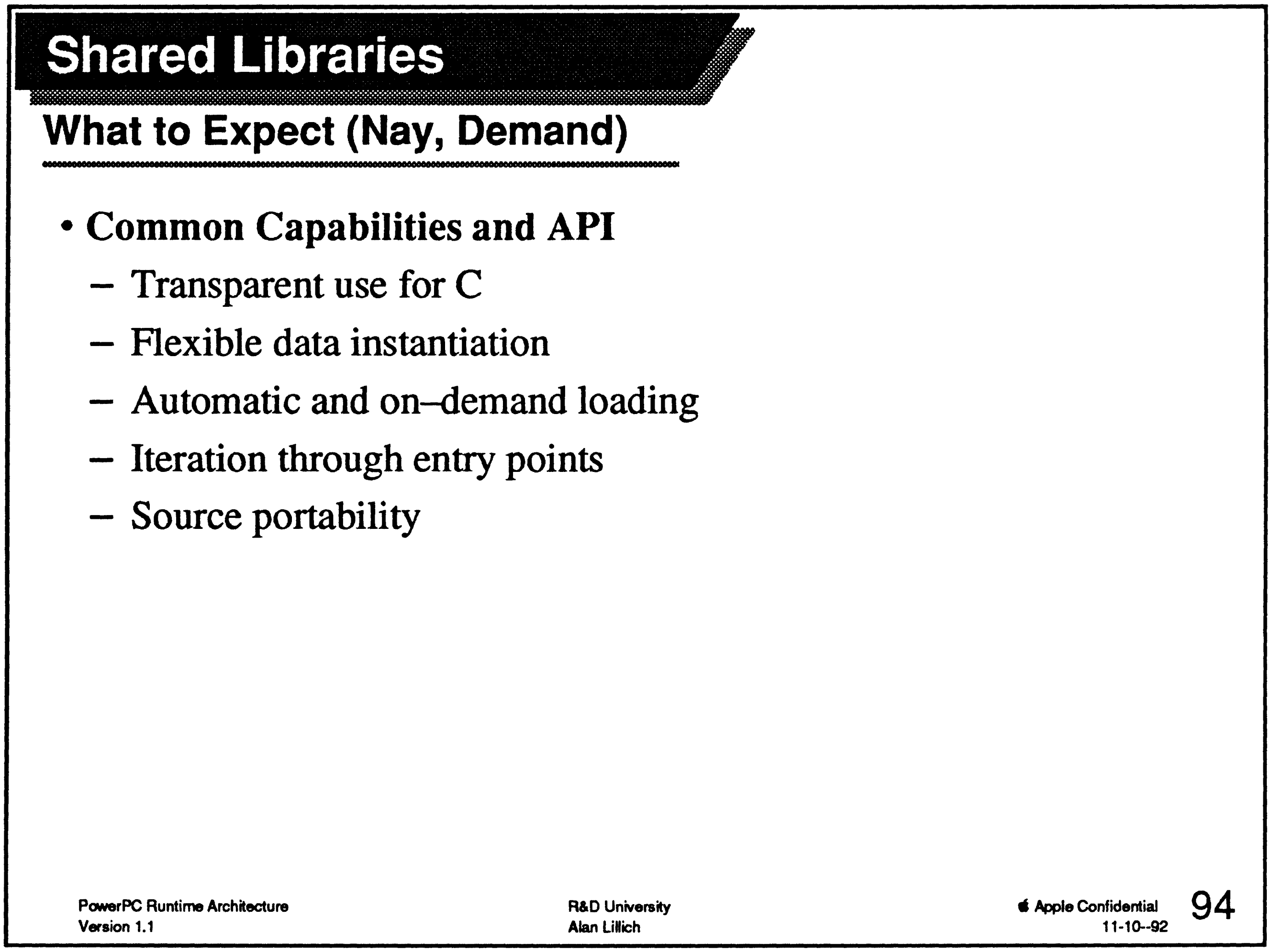
Status of SLM

- Hacked on top of 680x0 runtime
- Will layer on CFM for PowerPC
- Almost source transparent
- Additional tools to simplify C++ work
- Only supports system–wide data today

Shared Libraries

Status of Dinker

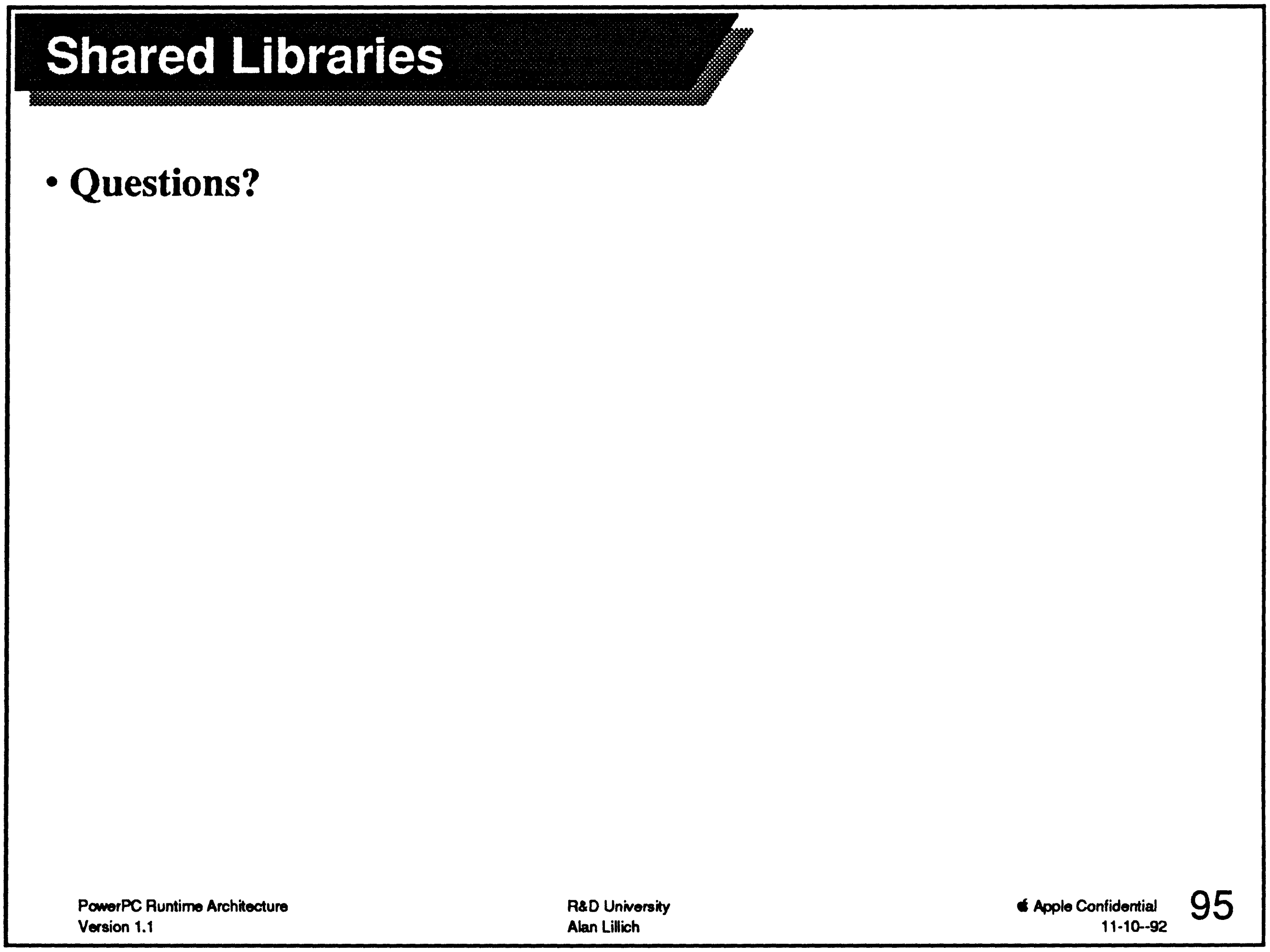
- Hacked on top 680x0 runtime
- Unsupported effort from ADG



Shared Libraries

What to Expect (Nay, Demand)

- **Common Capabilities and API**
  - Transparent use for C
  - Flexible data instantiation
  - Automatic and on–demand loading
  - Iteration through entry points
  - Source portability



Shared Libraries

- **Questions?**

System Software Issues

- Access to O/S and Toolbox
- Mixed Mode
- Micro–kernel

System Software Issues

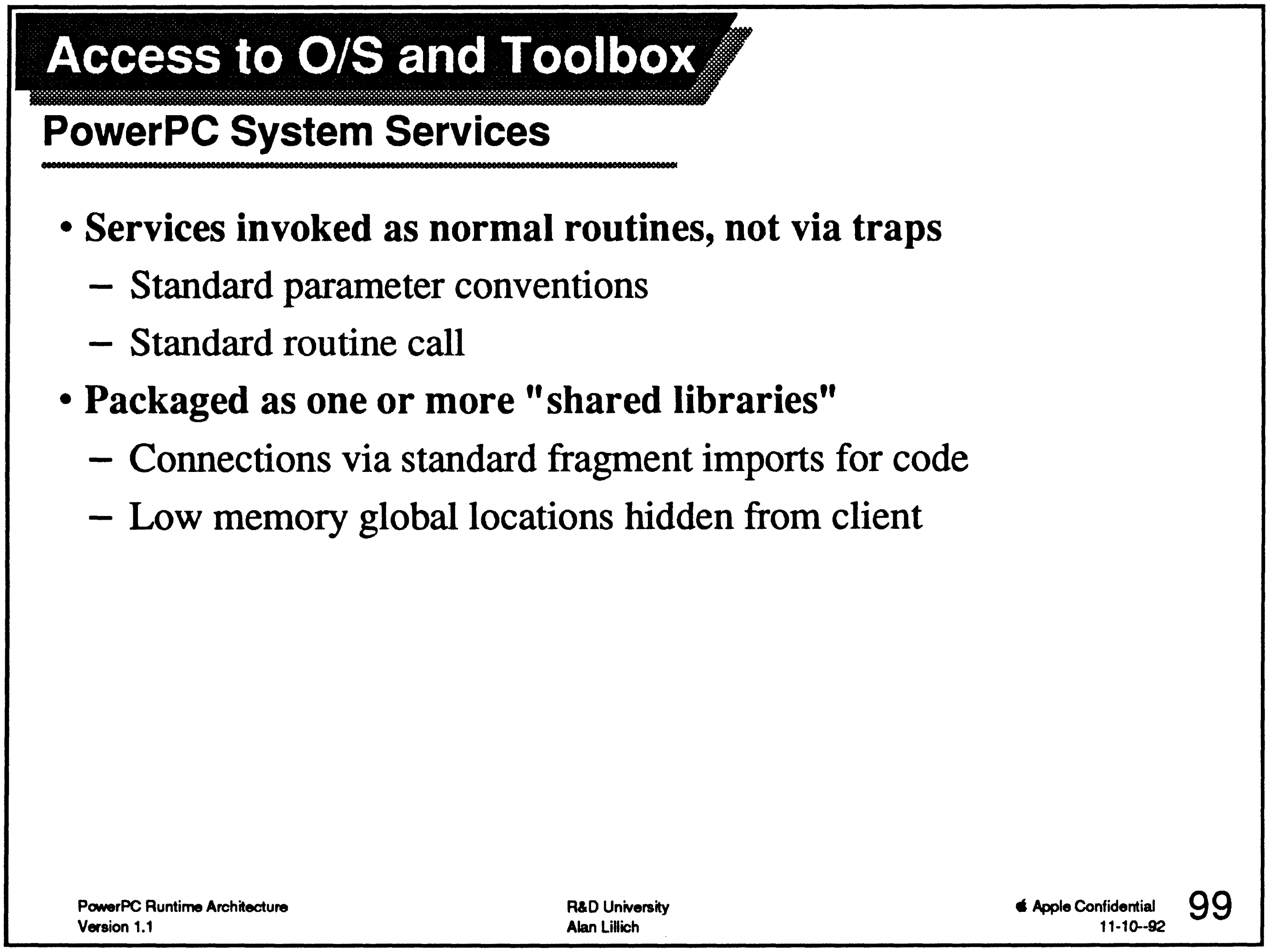
Access to O/S and Toolbox

- **“Old modified” API today for compatibility**
  - Only changes are for callbacks, due to mixed mode
  - Low memory globals still around and switched
- **New API coming for growth & evolution**
  - Will provide better error handling
  - Will allow transition to preemptive scheduling

Access to O/S and Toolbox

680x0 System Services

- **Parameter Passing**
  - Pascal conventions
  - Assembler conventions
- **Invocation**
  - A-Line trap, possibly with selector
  - Inline expansion, e.g. for low-mem "functions"
  - JSR to glue



Access to O/S and Toolbox

PowerPC System Services

- **Services invoked as normal routines, not via traps**
  - Standard parameter conventions
  - Standard routine call
- **Packaged as one or more "shared libraries"**
  - Connections via standard fragment imports for code
  - Low memory global locations hidden from client

System Software Issues

Mixed Mode

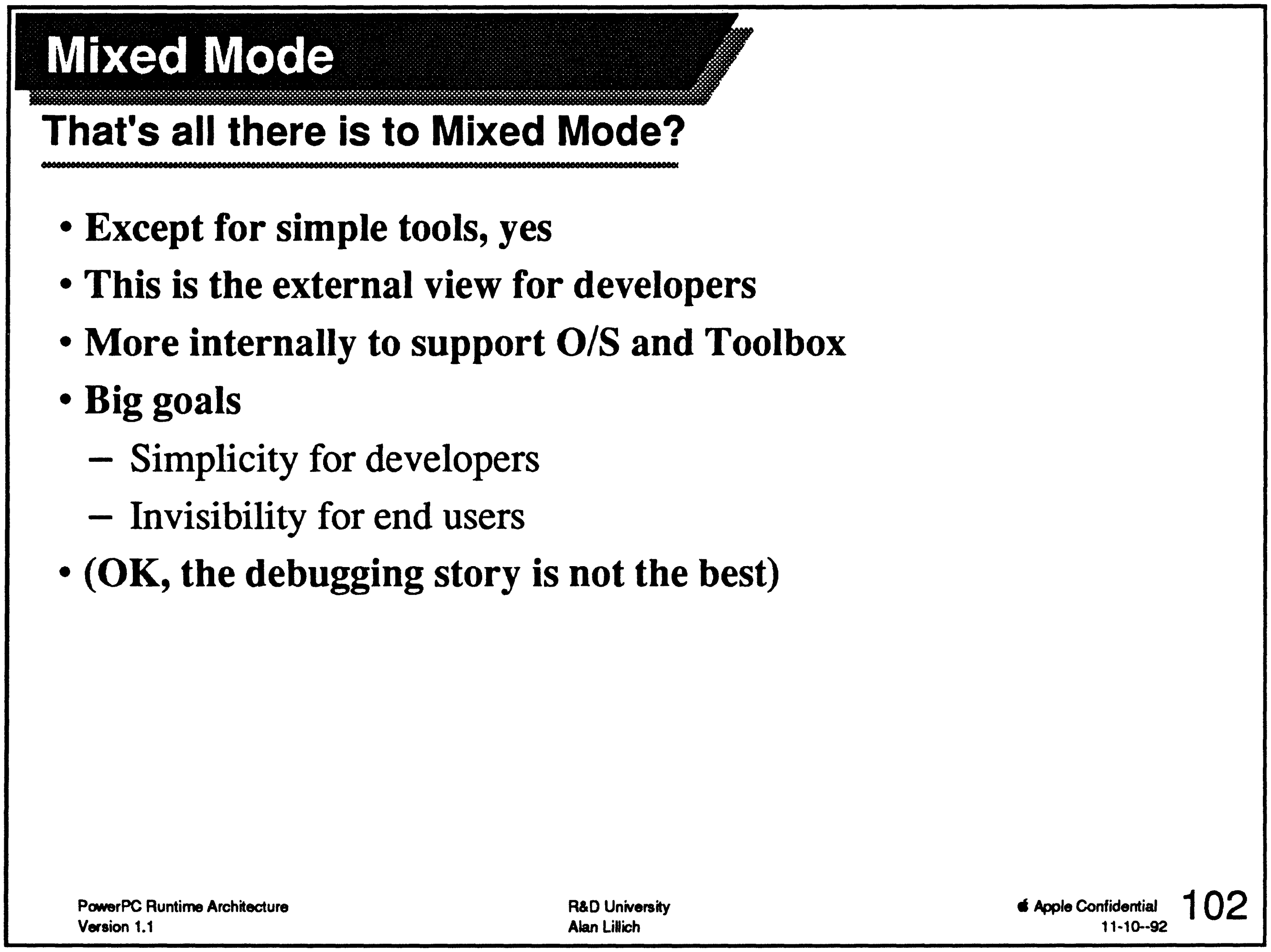
- **Allows mix of emulated and native code**
- **Static division at the fragment level**
  - "Fat" files and resources contain both forms
- **Dynamic division at the procedure call level**
  - 680x0 side may be ignorant
  - PowerPC side must be aware
- **API changes for callback pointers**
- **New service for callback invocation**

Mixed Mode

Routine Descriptors

- For flexibility use mixed mode descriptor pointers, not C routine pointers (to PowerPC Transition Vector)
- Dispatch service accepts calling info separately

goMixedMode	Illegal 68K instruction (e.g. A-Line trap)
selector	[execute, return, loadAndExecute]
version	0
executionMode	[codeType68K, codeTypePower, codeTypeEtc...]
procInfo	description of calling conventions
customParamProc	for custom calling conventions
theProc	ProcPtr to routine



Mixed Mode

That's all there is to Mixed Mode?

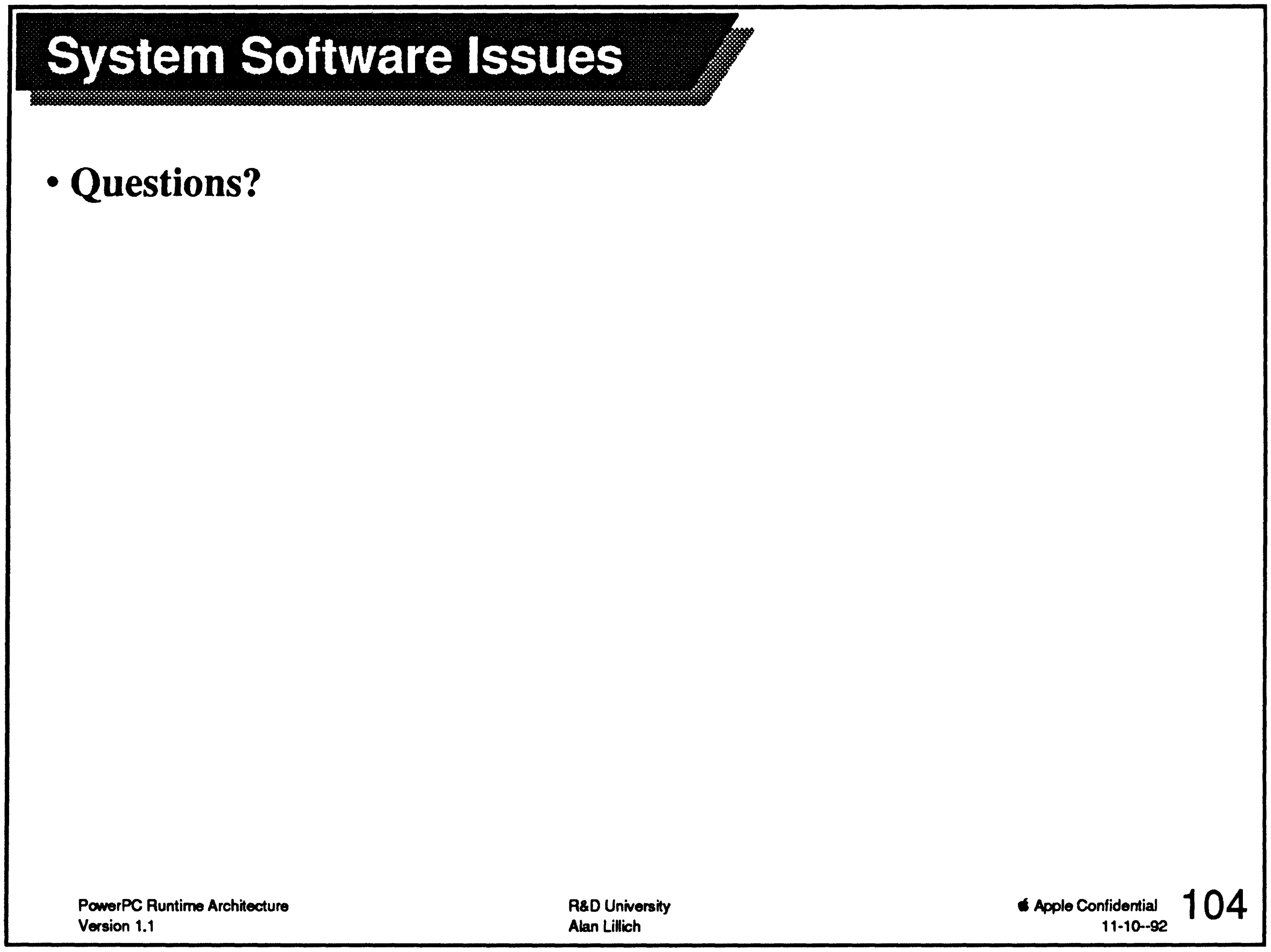
- Except for simple tools, yes
- This is the external view for developers
- More internally to support O/S and Toolbox
- Big goals
  - Simplicity for developers
  - Invisibility for end users
- (OK, the debugging story is not the best)

Eric Eidt for mixed mode interfaces

System Software Issues

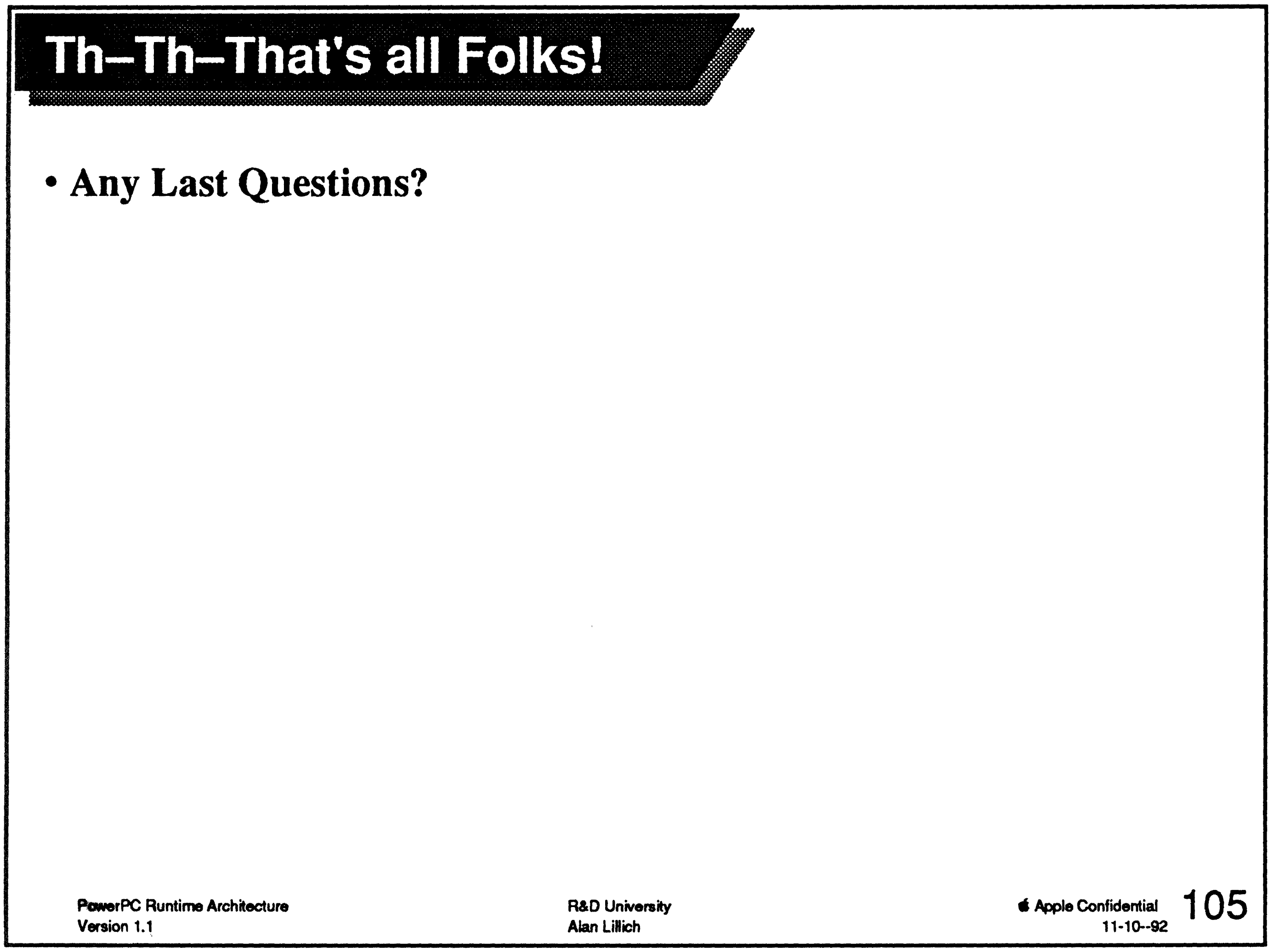
Micro–kernel

- **Won't be in first PowerPC release**
- **Features will appear over several releases**
- **Won't change low–level runtime model**
  - Nature of fragments (sections, exports, imports)
  - Global addressing (TOC, TVectors)
  - Calling conventions (parameter passing, register saving)
- **Will eventually change system software model**
  - First micro–kernel release supports faceless background tasks
    - Preemptively scheduled, probably just one address space
  - Toolbox will not be re–entrant & preemptive until later
    - Process Manager & cooperative switching will be with us for a while



System Software Issues

- **Questions?**



Th–Th–That's all Folks!

- Any Last Questions?

PowerPC Native Runtime Architecture

This is the specification of the native runtime architecture for Macintosh programs on the PowerPC. It deals only with the more primitive levels of the programming model. It specifically does not deal with the organization of the Toolbox, internals of the heap implementation, etc. This is written for a technical audience. Although written by the PowerPC Native Runtime team, the architecture is intended to be CPU neutral, presuming only a large flat address space and paged memory management.

Please send comments to Alan Lillich
AppleLink: A.Lillich
QuickMail: Development Tools:DSGMAIL
Telephone: 408-862-0029
Mail stop: 37-R

Revision history:

29-Sept-91	Initial release to PowerPC Runtime and DTE teams only.
04-Oct-91	Expanded in an attempt to at least note all important issues. Sorry, no change bars.
14-Oct-91	Reorganization of contents to focus more on specific native issues. Still no change bars.
11-Nov-91	Add details on stack frame, register usage, and parameter passing.
27-Jan-92	Add shared library information, improve illustrations, reformat slightly.
27-July-92	Major revision to improve content and style.

I. Overview

This note describes the runtime model for Macintosh programs on the PowerPC. It deals only with the more primitive levels of the programming model. It specifically does not deal with the organization of the Toolbox, internals of the various managers, etc. Our working definition of runtime architecture covers basic organization and protocols, not specific services. Of course services must be documented, but elsewhere than this architectural definition.

Although designed by the PowerPC Native Runtime team and containing PowerPC code fragments, the system architecture is intended to be portable to other processors. The design is biased towards systems with hardware memory management, ignoring the problems of real memory systems with small physical memory. This bias is most obvious in the removal of software controlled code segmentation.

This is intended as design documentation for the implementors of programming tools and system components such as the runtime loader. It may also serve as background for application and library authors. The information presented may be redundant for many, but this is written to be a standalone document. The material covered proceeds roughly from the more general to the more specific, from the larger grained to the smaller grained. The more general topics, such as use of address space, apply to user level code and not necessarily to privileged level code. More detailed topics, such as procedure calling conventions, apply to both.

This is a standard model, not a required model. It is sufficient for the needs of current procedural languages. It defines the interoperability model among languages and between user code and system services. Additional flexibility is designed into low levels of the implementation to support the internal needs of languages that require a different internal model. These areas are only briefly identified here. Full details are found in the documentation for the PowerPC Code Fragment Manager and Code Fragment Loader.

Much of the terminology and concepts come straight from the IBM implementation of AIX on the RS/6000. They have already developed adequate solutions to several problems. Following their lead also simplifies use of the AIX tools for initial prototyping and testing. We deviate from the AIX usage only where it is inadequate for our needs or where antique UNIX baggage can be disposed of at low cost.

The code fragments shown here are for illustration purposes and represent the PowerPC based Macintosh product. The actual sequences used may be slightly different. Compromises may also be made for higher compatibility with AIX to reduce time to market. Use of the AIX model will be necessary during the early development period.

<u>**Do not assume! This is not a minimalist port of the present 680x0 model.**</u>

II. Requirements & Goals

These are the project goals from the specific perspective of the Native Runtime Team. They do not minimize the work and responsibilities of other groups, but should maximize the overall group productivity and product quality.

Requirements are defined as those items for which slippage or alteration would have significant impact on the initial PowerPC products. Requirements would not be modified without visible warnings to project management. Goals are those items that we consider important but which may be modified over time for pragmatic reasons. Changes would be publicly discussed, but without the flares and flags of requirements changes.

Requirements:

- Ship product by the end of Q1 1994.

- Design a forward looking execution environment for native programs.
- Focus primarily on future hardware platforms, perhaps excluding today's low end.
- Allow reasonable source portability for well behaved code.

Goals:

- Be CPU and operating system kernel neutral.
- Minimize historical pollution.
- Reduce difficulties for other elements of the PowerPC Task Force.
- Maximize portability of existing Macintosh source and images.

Strategies:

- Make minimal assumptions about the O/S provided memory model.
- Provide fast register based subprogram calling conventions, used by all languages.
- Provide an easy to use and powerful shared library mechanism.
- Use subprogram calls to shared libraries for system services instead of a central dispatcher.
- Define full user access to CPU features, e.g. for arithmetic exception handlers.
- Propose a unified application software extension mechanism.
- Follow existing AIX standards where reasonable.

The first requirement is by far the most important, deserving of its separate listing. Our initial success or failure will be heavily judged by this although over the longer term other issues may gain importance. A primary tactic is to adopt existing RS/6000 AIX solutions where reasonable.

The next two requirements are closely related. The first is more of an internal issue. We must design a system that will carry Macintosh forward for another decade. The second is more external, recognizing that the base level of hardware will continue to improve rapidly.

Two key aspects of this forward view are provisions for shared code with dynamic binding and fast register based calling conventions. The first allows Apple and ISVs to deliver shareable images of common code to end users and have applications complete their binding to shared routines at load time. The second takes advantage of the large register sets found in modern CPUs to improve general performance. Combined with shared code and dynamic binding, fast calls allow dramatic improvement in system service response over a trap based mechanism. There is nothing to prevent the use of traps for system services, but they should not be necessary and we recommend that they not be used.

The last requirement is admittedly a declaration of freedom as much as a requirement. The source level portability of well behaved code is important, but most code should be dealing with internal algorithms or Toolbox. We will strive for portability but not enslave ourselves to 100% portability.

The first goal is almost a requirement, but downgraded to a goal since we won't be immediately proving our neutrality. Also, some of the architecture defined here is inherently CPU specific, for example the subprogram calling conventions. We intend to avoid concepts that could not be reasonably implemented on other platforms, especially 68030 based Macintoshes. (By which we mean 020 with PMMU, 030, and 040.) Being operating system kernel neutral emphasizes that the runtime architecture is largely layered on top of the kernel and should not make numerous or highly specific demands on the kernel.

The second goal complements the "forward looking" requirement. We should not carry forward all of the existing architecture simply because it exists. Pragmatic demands for backward compatibility may require some amount of historical pollution, but this should be isolated into specific toxic waste dumps with hopes of eventual cleanup.

The last two goals mainly say that this is not a research project and we should not fixate on an isolated perfection that risks global failure.

III. System Memory Model

Use of Address Space

The use of address space is quite simple, flexibility for the O/S kernel is maintained by making as few assumptions as possible. Programmers should expect to see a large flat address space with "register size" pointers. There may be a separate address space for each application, but this is not guaranteed. The address space will be large enough to give each application plenty of elbow room, even if the address space is shared. The runtime model does not presume that common structures are at the same logical address for all running applications as is done on many UNIX systems. Wild applications might clobber innocent bystanders.

The model is biased towards systems with virtual memory but does not require it. We intentionally ignore the problems of small real memory systems that led to the adoption of software based code segmentation in the current 680x0 model. We do not require that concurrently executing applications have independent address spaces, but we don't prevent that either. Applications should not make assumptions about whether memory is real or virtual nor about the number of address spaces. Some protection may be provided, e.g. that an application may not overwrite its code or that a section of memory belonging wholly to one application may not be read or written by another.

——— Explanatory Note ———

To date only 32-bit operation has been planned. Except when explicitly noted this paper in only discussing 32-bit operation. Although 64-bit implementations of the PowerPC are already underway, full 64-bit operation is not planned at this time. Much of the runtime architecture has obvious extension into a wholly 64-bit world, but no such operating system support is planned at present. Limited 64-bit operation will be allowed, this is described later.

The address space seen by an application is divided into large blocks of contiguous storage, called sections in this paper. A section of memory is characterized by:

- a base address and length
- whether it contains code, read-write data, or read-only data
- whether it is private or shareable
- the form and location of backing store (none, paging file, application file, etc.)

Shared memory sections are guaranteed to appear at the same logical location for all concurrent users. This allows the use of normal linked data structures and passing of pointers among clients. The operating system will choose the address at the time the shared section is created. If the contents of a shared section are to be saved to disk and later recreated it is the programmer's responsibility to preserve all embedded pointers. The operating system may allow you to request a specific address when creating a shared section, but such requests may fail even if that space is not in use locally.

An application may have separate sections for its code, static data, heap, and stack. All code will typically be in a single section, with all static data in another. The heap may actually be multiple discontiguous sections, new ones being added as existing ones become full. Ideally the heap and stack will be in separate sections, but the architecture does not mandate this.

An application may also utilize shared libraries having their own code and static data sections but sharing the application's heap and stack. These are described in more detail later. These shared libraries are identified when the application is linked and connected automatically when the application is launched. Their presence is generally invisible at the source code level. Resource and file based extensions are also supported, with even more power than in the present 680x0 model.

> —— Explanatory Note ——
>
> The shared libraries discussed in this paper are an intrinsic part of this software architecture. As such they will be provided first on the PowerPC and may never appear on 680x0 machines. They are particularly convenient for procedural languages like C, allowing a shared library to export both routines and data, and supporting flexible rules for the creation of instances of the "static data world". They do not provide explicit support for higher level notions such as the export of entire C++ classes as a single entity. A separate, higher level shared library system is under development that will provide explicit C++ support. This other system is less convenient to use for vanilla C and lacks some flexibility, particularly in regard to exporting data. This other system will be available on both PowerPC and 680x0 platforms. On the PowerPC it will be built on top of the base system.

Figure 1 shows a conceptual model of multiple address spaces. (With heavy emphasis on the conceptual!) The vertical orientation does not imply high or low address values, and the relative positioning of the sections is not significant. The horizontal orientation does imply equal logical addresses. This is significant only in the case of the shared code sections. Shared sections not "used" might still be accessible, depending on the complexity and tradeoffs of the virtual memory implementation. For example, user address space #2 might be able to see the code of library #3. In a single address space implementation the unshared sections of the the three user spaces may or may not be arbitrarily interleaved. There is no guarantee that the three portions would be segregated in private partitions, nor that they won't.

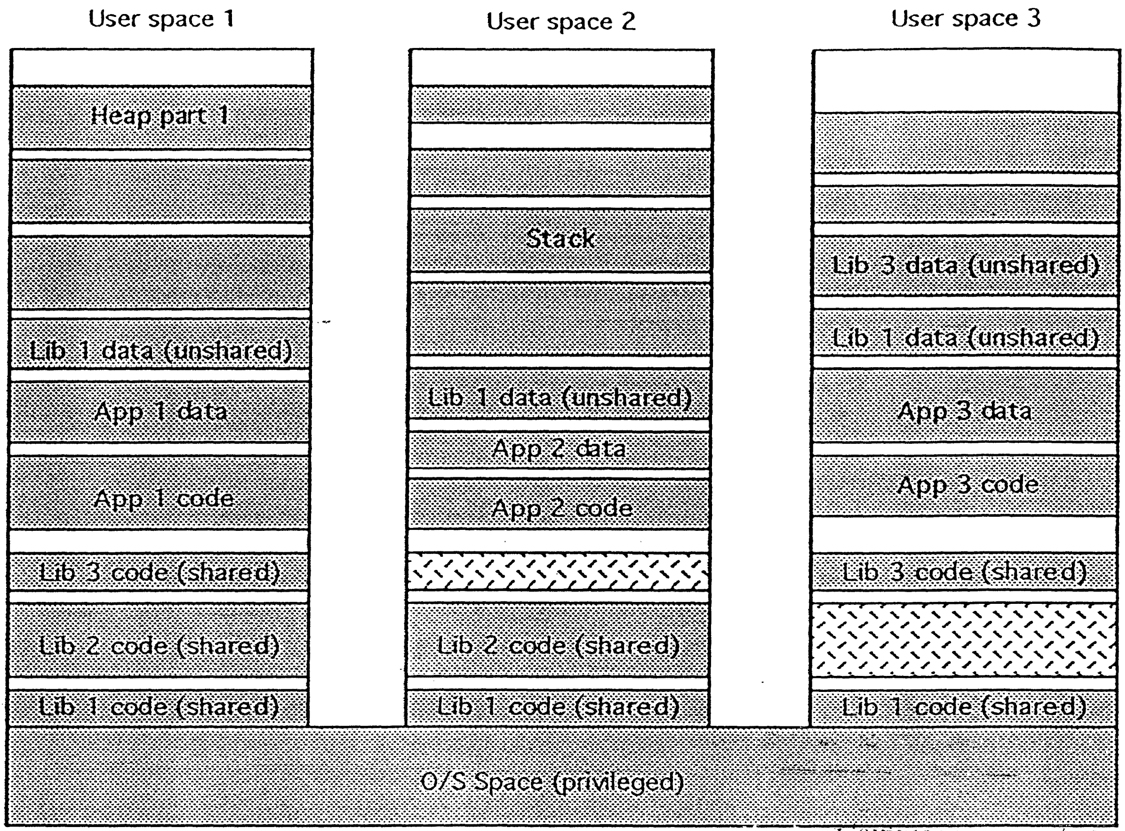


Figure 1. Conceptual model of multiple address spaces.

——— Explanatory Note ———

The existing Macintosh Memory Manager may not be modified to support the split, extensible heap model. The intent is to allow developer's to wean themselves from handle-based data. One possible implementation is to have one heap section reserved for the Memory Manager's handle-based allocation. NewPtr, malloc, and related routines would use other heap sections.

——— Explanatory Note ———

The bounded application partition of the current 680x0 model is deemphasized if not outright removed. The amount of code an application may have should not be bounded. The amount of static data, stack, and heap should not be bounded either. Whether they are bounded by the SIZE resource, whether the SIZE resource indicates an initial heap allocation, or whether the SIZE resource is removed remains a policy decision for human interface experts. The runtime architecture is neutral in this regard. Using the

SIZE resource to indicate the stack size and an initial heap size would allow a program to state minimal requirements on machines lacking virtual memory or having a limited System 7 style virtual memory capability. Should such machines exist.

——— Explanatory Note ———

We have consciously not defined a multi-section extensible stack model. To be done properly this requires hardware, operating system, compiler, and runtime support for the detection of overflow and "underflow" back across the boundary of stack sections. The lack of this capability is not seen as a major impediment to software development. With the removal of the Process Manager's partitioning and proper virtual memory support we will be able to efficiently handle very large stack sections. The virtual memory support is to allocate paging file space to the stack section only as needed.

——— Implementation Note ———

The initial PowerPC products may still have the System 7 Process Manager and virtual memory models. In this case the static data, stack, and heap may be bounded by the application partition's size and reside in one section of memory. The code may be mapped into a separate section and paged directly from the application file, but this is not guaranteed. If this mapping is not provided, the partition size will be automatically increased by the size of the code to account for the concurrent residence of all code. This should allow the same partition size to be tolerable for both 680x0 and PowerPC versions of an application.

——— Weasel Note ———

It may seem like few concrete statements are made about the use of address space. Everything from the System 7 Process Manager to fully separate address spaces can be defined to fit. To a certain extent this is a fair criticism. This is an architecture that must be implementable with several different O/S bases. The important concepts to walk away with are that code space is not bounded and managed in the ways that led to the segmentation model on the 680x0 and that shared structures are guaranteed to be at the same address for all clients.

Constraints and Caveats

The PowerPC hardware notion of segmentation is not presumed. A particular implementation may introduce hardware related limitations, but that is not part of the runtime architecture. For example, on the PowerPC a section might be limited to 256MB if the O/S does not support sections spanning hardware segments.

The maximum section size supported by the O/S must not be smaller than 16MB. This provides an arbitrary guarantee for a maximum size of code sections. Programmers should consider the use of shared libraries for larger amounts of code. The programming tools on each platform must in some way support code sections up to 16MB, they may optionally support larger sizes. A major factor is the displacement range for relative branches. Compilers may limit the amount of code that may be generated from one routine or source file. References outside of a source file should be assumed to be long and across code sections. Compilers and linkers should cooperate to optimize situations where such calls actually remain within a code section.

Of course a friendly operating system will support sections considerably larger than 16MB. This is particularly important for large data structures, which cannot be conveniently broken apart in the manner that code can with shared libraries.

A very important point is that code sections are intended to be used at a much coarser granularity than the present 680x0 segments. While virtual memory is not absolutely required, it is presumed to exist and code/constants should be paged directly from the executable file. Compilers, linkers, and loaders should preserve some notion of smaller scale segmentation both for source compatibility and control of locality. For example, the linker could arrange the code section by "segment" to improve locality of reference. The loader and O/S might use that information as a hint to improve paging performance.

This view of code sections is seen as a way to simplify application development. In exchange it requires that the memory management system perform in such a way that applications do not suffer from the lack of explicit segmentation. This implies a certain level of sophistication in the area of working set management and a certain minimum disk throughput. This approach is felt to have both high risk and high return.

IV. Executable Fragments.

Fundamental Properties of Fragments

One very important and universal concept of this model is the notion of execution units composed of blocks of code and associated static data. We've called these "fragments" to avoid conflict with other popular terms like module, object, and component. Everything that can be placed into memory and run is a fragment. Examples include applications, shared libraries, generic extensions like CDEFs, LDEFs, or WDEFs, and custom application extensions such as HyperCard XCMDs, Canvas tools, or 4th Dimension externals.

Fragments can be roughly divided into three classes, applications, shared libraries, and extensions. These classes are based on typical usage and properties beyond the notion of "fragmentness". Applications are things that can be launched to operate on documents. They may cooperate but are capable of independent operation. Shared libraries are special forms of extensions. Developers use them at link time to satisfy unresolved external symbols in the fragment being linked. Shared libraries are found automatically at runtime and connected to the fragment. Extensions are everything else. They are not identified at link time. They are explicitly looked up and connected at runtime.

Fragments have four fundamental properties:

- Loadable memory sections (code and static data),
- Exported symbols,
- Imported symbols, and
- Initialization, main, and termination routines.

——— Explanatory Note ———

Every fragment is a first class citizen, having all of the fundamental properties defined here. This is not to say that all fragments are created equal. Just as only police have the power of arrest and only the independently wealthy have the power of total leisure, some fragments have properties beyond the fundamental ones. Only applications have a SIZE resource defining stack and heap requirements. Only shared libraries are automatically connected at runtime. Discovering other usage examples are left as an exercise.

——— Explanatory Note ———

Three areas where we've gone beyond strict adoption of the AIX conventions are in support for multiple code and data sections, support for multiple container formats, and support for initialization and termination routines. These are all new in the runtime architecture defined by Apple. They are covered somewhat later in this paper and more fully in the documentation for the PowerPC Code Fragment Manager.

Code and Static Data Sections

Loadable memory sections are areas of code and static data that comprise the fragment's "executable image". In the prototypical case there is one section of code and one section of static data for each fragment. Code is not segmented as in the 680x0 case, we are not concerned about reducing the address space footprint. Large fragments are expected to be stored in the data fork of files, with virtual memory support for direct, read-only paging of the code. Shared libraries will typically have a separate copy of their static data for each concurrent use. This is invisible to the code, and is covered more in later sections.

——— Explanatory Note ———

The address space footprint of code is of less concern than in the present 680x0 model for two main reasons. First, we are presuming a large address space with respectable virtual memory support. The 680x0 model was designed to shoehorn programs into the very limited space on the original 128KB Macintosh. Second the PowerPC, along with the 68020 and better, has an unconditional branch with a decently long displacement.

——— Explanatory Note ———

The external storage of a fragment is known as a container. The runtime loader supports multiple container formats, as described later. Some formats, such as PEF and XCOFF use the term section internally, with some of their internal sections used only in support of runtime loading. These are not "loaded sections" as defined here.

Once loaded, the code and data of a fragment do not move around. The term loaded here means "prepared for execution", not simply "placed into memory". In the case of code bearing resources LoadResource simply places the contents into memory. The runtime loader prepares it for execution after it has been placed into memory and before it is used. If the resource is unlocked and moved the runtime loader must prepare it for execution again at its new location. The loader is not connected to the Resource Manager or Memory Manager. It will not automatically detect movement of a resource and reload the fragment. Unless otherwise noted, the term "load" means "prepare for execution" when applied to fragments, even when those fragments are the contents of resources.

——— Implementation Note ———

The Mixed Mode mechanism, used to support intermingled 680x0 emulation and native PowerPC execution, does provide some automation to allow old 680x0 code to properly use resources containing native PowerPC code. Programs compiled for the PowerPC should use the runtime loader to prepare the fragment whenever the resource contents are locked in memory. This has "zero" cost if the fragment is at the same location as the last time it was loaded.

The contents of code sections must be pure and position independent with regard to both code and static data. Pure code needs no modification to execute. (For example, standard 680x0 Macintosh code is pure. The implementation of "32-bit everything" utilizes impure code, having absolute data addresses that are relocated by the Segment Loader.) Pure code allows paging from application files, ROM execution, and makes code sharing easier. In this model on the PowerPC pure code is achieved by using self–relative branches within a fragment, indirect branches through pointers in static data between fragments, and addressing static data through a combination of a dedicated base register and indirection. The details of addressing are given in subsequent sections.

Position independence with regard to code means that the code will execute properly from any location in memory. It does not necessarily mean that the code can be moved in memory without modifying anything. It means that the code does not know or care where it is placed in memory. Generally this implies having no absolute code addresses within the code for branches to this or other fragments, case statement jump tables, etc. (Code can be pure and not position independent, as long as everything is placed at the right address.) We achieve position independence with regard to code on the PowerPC through the first two tactics for pure code, self–relative branches within a fragment and indirect branches through pointers in static data between fragments.

Position independence with regard to data means that the code will execute properly no matter where its static data is placed. It does not necessarily mean that the data can be moved in memory without modifying anything. It means that the code does not know or care where the data is placed in memory. Generally this implies having no absolute static data addresses within the code. We achieve position independence with regard to data on the PowerPC through the third tactic for pure code, a combination of a dedicated base register and indirection. Static data that is not directly addressable off of the base register is accessed indirectly through a pointer that is directly addressable.

——— Explanatory Note ———

> The combination of pure code and position independence, along with calling conventions for switching the dedicated base register make shared libraries "almost free". This model also supports multiple instantiations of fragments at the cost of just replicated static data. Consider writing a protocol handler in C with normal static variables maintaining the state of one connection. The handler can then be very easily instantiated once for each connection.

The architecture and runtime software support multiple sections of code and data. This is done to avoid designed–in limitations for "nontraditional" languages and unanticipated future use. The use of a single code section and a single data section is adequate for today's common procedural languages. That is the standard output model of the linker. Again, multiple code sections are not intended to be used in the same fine grained manner as 680x0 code segments.

Exports and Imports

A fragment may export symbols defined within any of its sections for use by other fragments. These are exported by name. The names of the global symbols to be exported must be explicitly given to the linker when the fragment is linked. Exports from shared libraries are used during the linking process to resolve undefined external symbols into imports and again during the loading process to resolve imports into addresses.

After loading the locations of exports from a fragment may also be determined by query using the export's name. This is particularly useful for extensions that are loaded on explicit request. They may export an entire API.

——— **Explanatory Note** ———

Although code symbols may be exported, the addressing model is such that only data symbols are normally exported. Routines are exported through a descriptor in the data. There are conventions for compilers that make this transparent to programmers. A routine named "foo" in the source should have a code label of ".foo", and a descriptor named "foo". Telling the linker to export "foo" then does the right thing. Global data items keep their source names.

Imports are code and data items that a fragment requires from other fragments. They are denoted with a library–name/symbol–name pair. Programmers present shared libraries to the linker just as they present traditional linker libraries today. The symbols exported from the shared library become available during the link like ordinary global symbols. When the linker resolves an undefined external symbol to an exported symbol it will record that as an import in the new fragment rather than actually copying the referenced code or data.

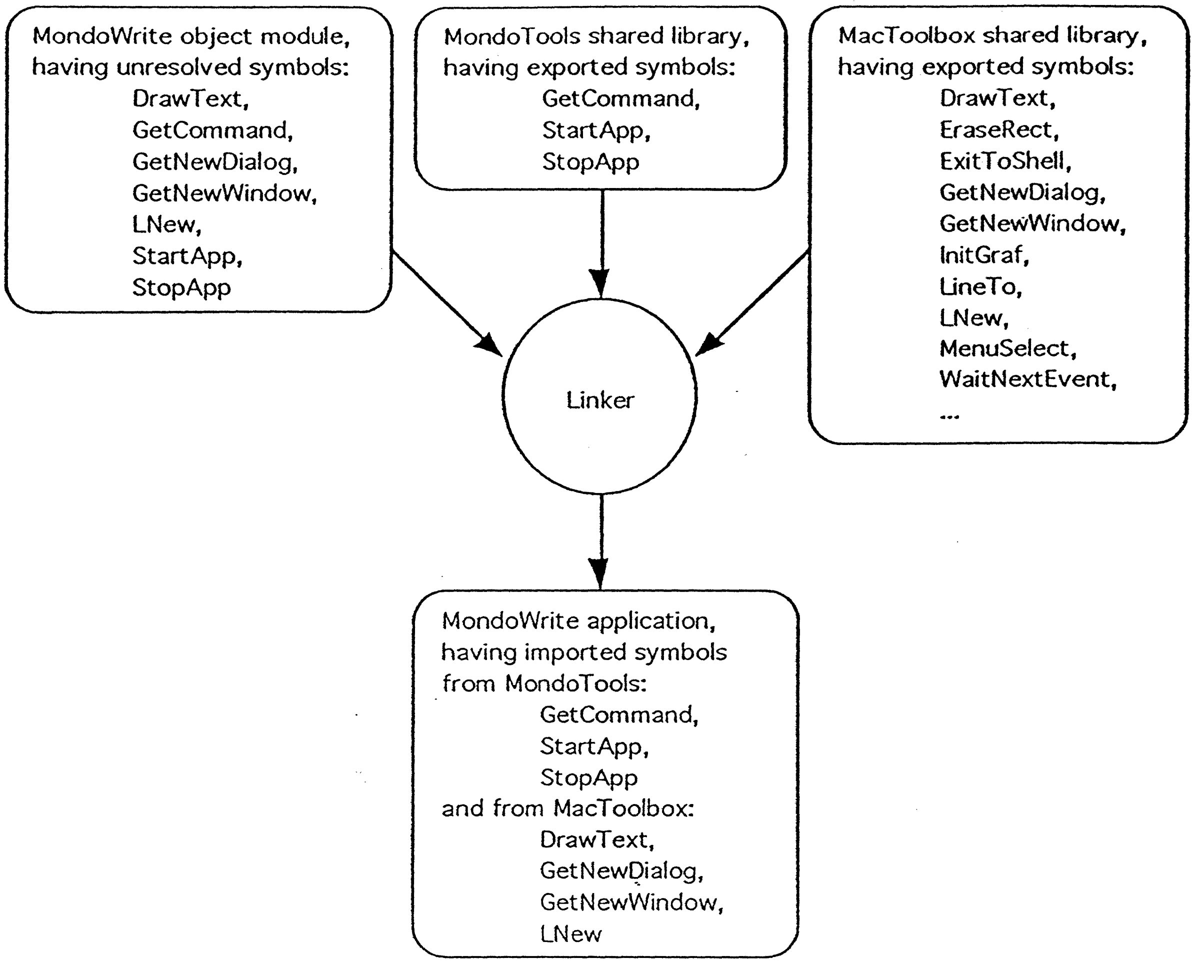


Figure 2. An illustration of export and import handling by the linker.

Figure 2 is an illustration of the linking process including resolution of undefined external symbols to imports from shared libraries. It is worth noting that the shared library used for linking need not be an actual implementation. The link time library need only export the right names and contain appropriate version information. This can simplify development considerably. A dummy linking shared library can be created as soon as an API is defined, allowing coding and linking of clients to proceed. Implementations are only needed to test the client, these could be evolving versions supplying partial functionality.

The imports are automatically resolved during loading to exports from shared library fragments, providing the actual runtime address. The imported library will itself be loaded if necessary. The runtime resolution must be to the "same" library as the link time resolution. The runtime library must have the same name and compatible version numbers. This is described more in a later section on shared libraries. The code generation details related to the use of imported symbols are given in later sections dealing with the addressing of global data and procedure calling conventions.

——— Explanatory Note ———

This approach is probably better called "dynamic binding" than "dynamic linking". It is very important to realize that all symbols must be resolved at link time. Those symbols that are resolved to exports from a shared library become imports and have their actual addresses bound at runtime. Each import specifies both the symbol name resolved by the linker and the name of the shared library in which it was found. Each imported symbol is looked up in the same library at runtime.

——— Implementation Note ———

The precise definition of a "shared library" is expected to evolve as tools and runtime software become more sophisticated. Initially shared libraries may be files of type "shlb", whose name is taken as the simple file name. They will be found at runtime through a simple search path including at least the application folder and system extensions folder. Special libraries such as a ROM-based toolbox would be specially known. It is desirable to move as soon as possible to a general registration scheme, allowing libraries to be found anywhere in the network.

Figure 3 is an illustration of application, shared library, and extension fragments in operation. (As with figure 1, heavy emphasis on illustration!) MondoWrite and MondoDraw are applications. MondoTools and MacToolbox are shared libraries. MondoTools is a "normal" shared library, having a separate static data instance for each application. MacToolbox is "abnormal", having global shared static data. MondoLDEF is an LDEF resource.

The solid arrows represent imports, for example MondoWrite imports from MondoTools and MacToolbox, while MondoTools imports only from MacToolbox. The names by the arrows are the symbols being imported. They must of course be exports from the library at the arrowhead.

The dashed line from MondoWrite to MondoLDEF signifies that MondoWrite does not access the LDEF as an import. It must load the resource (in both senses), and obtain a procedure pointer to the LDEF. This could either be via an agreed upon export name or via the main routine of the LDEF fragment. The LDEF itself does have imports just like any fragment. They are automatically resolved when the fragment is loaded.

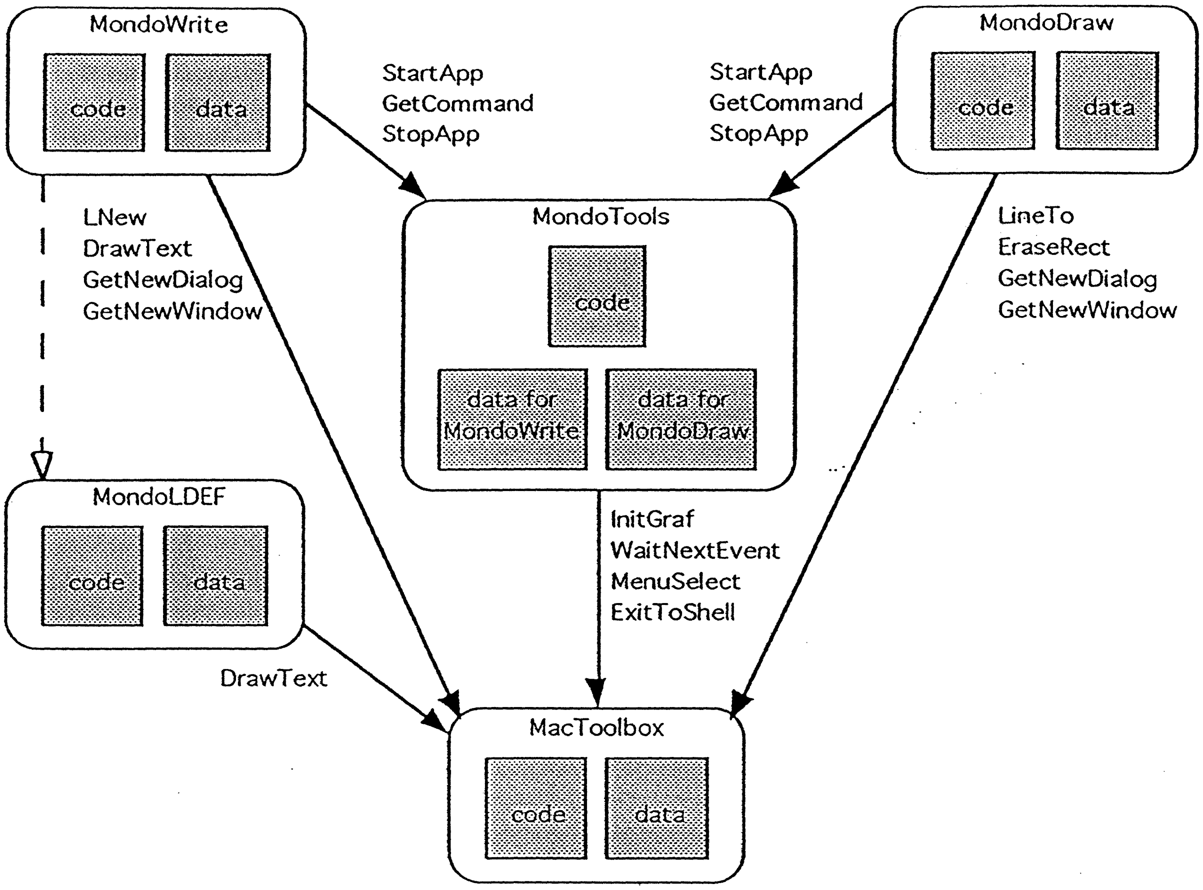


Figure 3. An illustration of application, shared library, and extension fragments in operation.

Initialization, Main, and Termination Routines

The remaining basic capabilities are the initialization, main, and termination routines. Every fragment may define these three routines, separate from its list of exports. The initialization and termination routines may be left undefined for any fragment. Applications must have a main routine, others need not. These rules are generalities based on typical usage, other specific uses may have their own specific requirements.

The initialization routine is called as part of the loading process. It provides a place for specific language runtime support or even hand written code to perform initialization before the fragment is considered fully loaded. Integration of language initialization code and hand written code into a single call is to be defined by compiler vendors and perhaps supported by linkers. The termination routine provides the inverse, allowing cleanup before unloading.

The interpretation of a main routine depends on the use of the fragment. For applications it is the usual main entry point. For shared libraries it is ignored. For extensions with a single entry it could be used instead of an export to avoid standardizing on a particular name. The main routine can be found by query after a fragment is loaded.

When the loading of one fragment causes a (currently unloaded) shared library to be loaded (to resolve imports), the initialization routine of the new shared library is called before that of the fragment. This allows the fragment's initialization to utilize the shared library. Mutually dependent libraries may specify which must be initialized first. Termination routines are called in the inverse order of initialization routines. This process is described more in a later section.

The initialization and termination routines provide significant power. They are crucial in supporting the full use of C++, allowing any fragment to contain static objects having constructors and destructors. The also allow the creation of self-initializing managers. Combined with initialization ordering, suites of managers can be self-initializing in the proper order.

Fragment Storage

The external storage of a fragment is called a container. A container may exist as any contiguous piece of storage, such as the data fork of a file (or a portion thereof), in ROM, or the contents of a resource. The notion of container is separate from a fragment to clearly separate the logical properties of fragments from the physical organization of containers.

Fragments are managed at runtime by the PowerPC Code Fragment Manager (CFM) and Code Fragment Loader (CFL). The CFM does the high level work, knowing what is loaded, creating the code and data sections, connecting imports and exports, etc. It operates transparently, much like the 680x0 Segment Loader. The CFM also supplies an API allowing fragments to loaded on explicit request. This API is defined elsewhere. The CFM does not supplant services like the QuickTime Component Manager. The Component Manager finds a container, the CFM prepares it for execution.

——— Explanatory Note ———

The general term "runtime loader" used earlier refers to the CFM, not the CFL.

The CFL provides low level services, mainly to the CFM. Its API provides routines and a loading protocol, shielding the CFM from the physical format of the container. Following the loading protocol, the CFM asks the CFL about the number of memory sections and their characteristics, the exported symbols, the imported libraries and symbols, etc. The CFM tells the CFL where the memory sections are placed, the addresses of imported symbols, and when to perform relocations. The CFL performs the actual relocations.

The CFL design includes support for multiple container formats and loader implementations. It will find the appropriate loader for a container at runtime. Standard implementations include XCOFF and PEF. The CFL is intended provide adequate support to integrate languages such as Lisp or Dylan with significantly different internal runtime models than C. By utilizing a Dylan container and loader, shared libraries written in Dylan could transparently interoperate with "normal" languages. The CFL API is defined elsewhere.

—— Explanatory Note ——

The CFL API is available to support the implementation of alternate loaders. Except in very restricted circumstances developers should call the CFM for loading services and not the CFL. The CFM implements the semantics of fragments, the CFL allows it to do so without knowing the external storage format.

—— Open Issue ——

The exact means by which loaders are found at runtime is somewhat open. Ideally a registration scheme would be available. Initially a file scan at boot time may be used.

XCOFF is the standard object and executable format used in AIX and PowerOpen. While an XCOFF loader is provided, arbitrary UNIX programs and libraries are not guaranteed to be usable under the Macintosh O/S. Use of UNIX memory or process services, dependence on the AIX memory model, etc., will cause problems.

PEF is the PowerPC Executable Format for Macintosh. It provides a dramatically smaller container than XCOFF, mainly in the representation of the load time relocations. This reduces both disk usage and load time. PEF is a format for executable containers only, not a relocatable object module format. Details of PEF are documented elsewhere.

—— Implementation Note ——

The IBM linker and initial Apple linker will take XCOFF as input and produce XCOFF as output. A conversion tool exists to convert linked XCOFF into PEF. Later versions of the Apple linker may produce PEF output directly.

V. Global Addressing, the TOC.

This section discusses the standard model for the addressing of global code and data. This includes how one fragment accesses its own static data and how it accesses the code and static data of other fragments. The dreaded TOC will be introduced, demystified, and vanquished.

——— Motivational Note ———

The PowerPC along with other RISC machines has a load/store architecture and fixed length instructions. One effect of this is that immediate displacements for addressing are limited in size, 16 bits in the case of the PowerPC. Loading a value at an arbitrary 32-bit offset from a base register requires at least two instructions on the PowerPC, three instructions and an additional register if you want to save the address. Constructing a pointer takes two instructions.

Using 32-bit offsets for static data would either require multiple instructions for all static data references or compile time segregation of static data items into 16-bit and 32-bit pools, with the associated risk of 16-bit pool overflow at link time.

Another approach is to address static data items indirectly. A dedicated base register is kept to a pool of pointers. A 16-bit offset allows 16K pointers, which means 16K individual items addressable from one place regardless of their aggregate size. Loading an item takes at most two instructions, obtaining a pointer takes one instruction.

Frequently used items should have their pointers kept in a register as part of a compiler's standard common subexpression and register allocation processing. A read, modify, and write cycle takes just one extra instruction to load the pointer.

This approach has the further benefit of implicitly supporting address independence for individual static data items. This allows shared libraries written in such deficient languages as C to transparently export and import static data. (C is a problem because you cannot tell at compile time whether a particular external variable will belong to you or someone else.) Static data items known to belong to a single unit, such as those with the C keyword "static", can be grouped together and share a single pointer.

The indirect addressing approach is the one used here. Yes, the two instruction case does involve two data memory references, one for the pointer and one for the value. But a good compiler and decent coding habits will reduce the loading of pointers. In return we gain significant expressive power and growth space for larger programs.

This model is taken directly from IBM's RS/6000 AIX implementation. It works. We have defined some details from a different view, but the basics are unchanged.

We'll first present the TOC in the framework of typical C usage, taking advantage of concrete examples. We'll then provide the formal notions and generalized nature of the TOC.

The TOC Nextdoor

The addressing of all static data and routines across fragment boundaries is done indirectly, through pointers kept in an area known as the TOC. Although an acronym for "Table Of Contents", the TOC is more like a personal address book. It belongs to a fragment and tells that fragment where other bits of the world are. The TOC does not tell the outside world where to find things within its own fragment. The TOC does tell a fragment where its own static data is. This allows code to be shared among multiple clients, each having their own copy of the TOC and static data.

——— Explanatory Note ———

The first sentence in the above paragraph should be parsed as

"... (all static data) and (routines across fragment boundaries) ..."

not as

"... all (static data and routines) across fragment boundaries ...".

This should become clear later in this section.

We're talking here only about "direct" source references, that is source that does not explicitly involve a pointer. The explicit use of pointers, whether for data or routines, does not involve the TOC. (Except perhaps to obtain the pointer value if it is itself a static variable.) The initialization of the pointer is presumed to be done correctly. This may involve static initialization with relocation by the runtime loader or code actually copying a pointer from the TOC.

——— Explanatory Note ———

For example, consider this (contrived and useless) C code:

```
extern int foo;
extern int *bar = &foo;

foo = foo + 1;
*bar = *bar + 1;
```

The use of foo is direct at the source level, bar is not. Assuming the TOC pointer to the value of bar is already in a register, both assignments generate the same instructions. The first uses the implicit TOC pointer associated with foo in this unit. The second uses the explicit storage of bar for the pointer. The initialization of the TOC pointer for foo and for bar itself are accomplished in the same way.

Add a picture here.

On the PowerPC, general purpose register 2 is dedicated to point to the TOC. It is commonly called RTOC. The contents of this register are saved, modified, and restored for calls across fragments. Every routine assumes that RTOC is pointing to its TOC upon entry. The details of this are presented later with other calling conventions.

The TOC is just a portion of the static data section described earlier. Whether before, after, or in the midst of programmer declared static data does not really matter except as it affects the performance of runtime loading. For now the TOC is simply somewhere in the data section of a fragment and GPR2 is loaded with its address before entering any routine in the fragment.

——— Explanatory Note ———

A reminder that this section presents the TOC in its standard C usage. The actual rules are somewhat more general and presented later.

The pointers in the TOC are allocated by both the compiler and linker, never at runtime. The TOC entries are filled in with addresses during runtime loading; new entries are not allocated at runtime. Code is pure and ROM-ready after linking, all offsets to TOC pointers are fixed. Runtime loading may also involve the relocation of pointers outside of the TOC. One example was given earlier, others will be covered later.

The C language has two classes of static data, "extern" variables visible across compilations and "static" variables visible only within a compilation. If neither keyword is used for data outside of a procedure, "extern" is presumed.

Keyword "static" variables are known at compile time to belong to the fragment of the source being compiled. All "static" variables in a compilation are placed contiguously in the static data section by the compiler and accessed through a single TOC pointer. The compiler allocates the TOC pointer. References to a "static" variable first load the common TOC pointer then load at the appropriate offset from it.

For "extern" variables the compiler does not generally know which fragment will contain the storage. It may end up in the fragment being compiled or it may end up coming from a shared library. Each "extern" variable is addressed through a separate TOC pointer. These pointers are also allocated by the compiler. References to "extern" variables of course first load the pointer from the TOC then the value of the variable.

——— Picture Place ———

Put in a picture showing a simple C source, generated code, and storage diagram. Use dis output to illustrate assembly programming at the same time.

——— Programming Note ———

By using "static" instead of "extern" where possible programmers give the compiler a better chance to keep a variable's TOC pointer in a register. This allows later references to be done with a single instruction.

Assembly language programmers may utilize whatever clustering is appropriate. As always, they must do manually everything that compilers do. In this case that means creating TOC pointers explicitly in their source.

——— Explanatory Note ———

The pointers in the TOC are to the actual referenced items. This includes all TOC pointers, whether to a fragment's own data or data imported from another fragment. There has been some confusion in the past that the TOC pointer for an import pointed to the TOC pointer in the owning fragment, that in turn pointed to the imported item. (As though the import had a handle and the owner had a pointer.) This is not true. All TOC pointers point directly to the referenced item. It becomes obvious when you note that the compiler generates one form of code not knowing whether the final reference will be an import or to "owned" data. The linker decides that.

A similar situation occurs for "extern" routines. The compiler does not know whether they will end up being in the same fragment or will involve a cross-fragment call. The details are more complex than for data, with the compiler doing a little of the work and the linker doing a larger part. The details are given in a later section dealing with calling conventions. For now it suffices to say that the linker allocates a TOC pointer in the calling fragment, and code in the calling fragment that uses that pointer to complete the call.

——— Anticipatory Note ———

An earlier note mentioned that code symbols are not usually exported. Each externally visible routine has a descriptor associated with it. The descriptor contains the code address for the routine and the address of the TOC for the routine. The TOC pointer involved in cross-fragment calls points to the descriptor. Each caller has a pointer to the descriptor, just like references to any external variable. Glue code involved in cross-fragment calls uses this TOC pointer. This will all be detailed in a later section on calling conventions.

In summary, a TOC contains:

- One pointer for each "extern" data item used by the fragment.
- One pointer for the "static" pool of each compilation unit in the fragment.
- One pointer for each imported routine called by the fragment.
- Actual static variables if a compiler were to put any there.
- Anything an assembly language programmer tosses in.

The linker performs the final allocation of TOC space and optimizes the use of TOC elements. Compilers allocate TOC pointers symbolically to "data item foo", "routine descriptor bar", etc. When combining separately compiled units into a fragment the linker will recognize common references and merge the TOC pointers. Each TOC will normally have just one pointer to any given routine or data item.

——— Implementation Note ———

The linkers (IBM's and Apple's) only combine TOC entries whose name and contents both match. Using IBM's assembler syntax, a TOC pointer to a static variable named foo could be declared via:

```
label:  .tc       foo[tc], foo[rw]
```

The label is strictly for local reference to the pointer. The name of the TOC entry is "foo[tc]", its contents is the address of "foo[rw]". The name and contents are used by the linker in merging TOC entries, the label is not. Compilers of course always generate TOC entries in the same way, ensuring that they will be folded. Assembly language programmers can follow the same rules, given in a later section.

The Zen of TOC

No, Zen–o–TOC is not a new candy bar. Let's put on the rose colored glasses and meditate.

- The TOC is a collection of code generation and system software conventions.
- The TOC is the static storage to which RTOC points on entry to a routine.
- The TOC is the static storage which can be addressed in one instruction.
- The TOC is that which provides a dynamically bound gateway to the outside world.
- The TOC is that which provides a dynamically bound gateway to non-TOC static data.
- The TOC is what the 680x0 A5 world should have been.

The TOC is not an inviolate barrier. There are no TOC Police at compile time, link time, or runtime. There are a set of code generation and system software conventions that support reasonably efficient pure code, dynamic binding, and componentized software construction. It has particular value on machines like the PowerPC that lack memory reference instructions with large displacements.

The TOC is there to let routines find the static data and external routines they are interested in, where that interest is a direct reference in the source. The compilers, linker, and system software ~~conspire~~ cooperate to add a level of indirection in these cases, providing wondrous benefits to programmers without extra (visible) cost. As mentioned earlier, explicit use of pointers need not and should not go through the TOC.

——— Explanatory Note ———

The compilers mentioned here are those for conventional procedure languages. Other languages, particularly dynamic languages such as LISP, are free to use other conventions internally. These conventions are necessary for interaction with system software and allow transparent mixing of code from compliant languages. You could implement a shared library in one language today and another language tomorrow with no impact on its clients.

TOCs are really properties of routines. When calling into a fragment the new TOC value is known separately for each routine. The routine descriptors mentioned above provide this. There is no constraint that all routines in one fragment share the same TOC. i.e. that the TOC value in all of their descriptors be the same. This is a decision between compilers and linkers. There is no constraint that a fragment have a single data section with the TOC inside it. A linker is free to create code and data sections in any way it chooses, placing the TOC in any data section. The only constraint on the linker is that it not violate the legitimate assumptions of compiled code.

—— Implementation Note ——

The general nature of these legitimate assumptions should be clear from this paper. (If not the paper is deficient.) They are implicitly presented in various rules and examples. They include such notions as the declaration and use of TOC pointers, the initialization of non-TOC pointers in the static data, linker processing of calls across compilation units, etc. Their details are to be found in documentation on the linker's input and internal processing.

—— Implementation Note ——

Static data sections of fragments have an attribute that describes a granularity of sharing for them. This is discussed in a later section. Note that should a linker be implemented to create multiple data sections with different levels of sharing, the TOC should be placed in the least sharable, perhaps in a section by itself. The TOC should be replicated at least as often as anything else since it contains the pointers to address the other sections.

Compilers decide whether particular calls within a source file are strictly local or might possibly involve a TOC switch. Compilers provide with each compilation a description of the individual TOC pointers necessary for that unit. Compilers do not create a "TOC area", they do not deal with relationships between TOC pointers. The linker makes the final determination of whether potentially switching calls actually do switch. The linker makes the final decision of where individual pointers appear in the TOC. The linker makes the final decision about collapsing identical TOC pointers.

Compilers for various languages may all have their own rules for the creation of TOC pointers, association of multiple variables with one TOC pointer, switching of the TOC on particular calls, etc. Languages with a module/package/unit concept and hierarchical separate compilation in particular may have more stringent rules than C. (Hierarchical separate compilation allows nested procedures to be separately compiled while keeping their nested context.)

So far everything in the TOC has been a pointer. There is no constraint on the TOC to contain only pointers. Compilers and assembly language programmers are free to place any static data in the TOC. This view is encouraged by the first two bullets above, i.e. that the TOC is the "easy access static data".

There are of course linguistic and usage constraints on placing data directly in the TOC. The compiler may only place directly in the TOC items that it knows will be owned by the unit being compiled. The cases in which placement of variables in the TOC is profitable are not obvious. The decisions require the same kind of usage information necessary for register allocation.

For example, C items declared with the "static" keyword have just file scope and could always be placed in the TOC. But if the compiler uses a single TOC pointer for all statics in a compilation, this pointer may end up in a register anyway, giving easy access to all "static" variables.

Items declared with the "extern" keyword have global scope and could only be placed in the TOC if the source contained the defining occurrence. If the unit being compiled does not have a defining occurrence it is possible that the variable will end up coming from a shared library. It would be perfectly fine for the unit with the defining occurrence to place the storage in its TOC and export the variable. Other units would reference the variable through a TOC pointer as usual. This might even include separately compiled units linked into the same fragment.

Putting a frequently used item directly in the TOC might benefit the owner of the data by removing the indirection for loads and stores. But with a large register set the TOC pointer for that frequently used item might stay in a register anyway, an optimization available to all users of the data. The big beneficiary of allocation in the TOC is probably a frequently called routine with few references to the static data. They would save the load of the TOC pointer on each pass through the routine.

A variable's size also plays a role in deciding where to allocate it. A single item of four bytes or less will take no more space in the TOC than the pointer. Larger items use up TOC space faster, increasing the risk of TOC overflow. C "static" versus "extern" are different here too. The former share a pointer, the latter have one pointer per item.

This is clearly an area ripe with opportunity for compiler optimization and pragmas.

——— Implementation Note ———

Neither the IBM nor Apple compilers put data directly in the TOC yet.

It is also desirable to reserve some space in all TOCs at fixed offsets for use by system software. This provides a place to store context that should be associated with fragments and switched cheaply with cross-fragment calls. We are proposing that 64 bytes (16 pointers worth) be reserved at offsets 0 to 63. No specific use for this space is defined yet. This space is reserved for use by Apple.

——— Open Issue ———

Some space will be reserved, what is proposed is that the exact amount be 64 bytes. We have arbitrarily proposed that 64 bytes at offsets zero to 63 be reserved. Nothing has been decided about use of this space by the Resource Manager or other components of system software.

——— Rationale Note ———

The 680x0 use of 0(A5) as the QuickDraw globals pointer springs to mind, but that is actually a poor example. There is one set of QuickDraw globals per application. This pointer should be kept as a static variable in a Toolbox shared library, with one instance of the library's static data for each application.

A more interesting possibility occurs for the Resource Manager, introduced because of shared libraries. This concept was first proposed in the Dinker design by Jed Harris. There is currently one resource chain per application. With the advent of shared libraries it becomes very desirable to have one resource chain per shared library, with automatic switching during procedure calls. Consider an application using several shared libraries, each with a DLOG resource number 123, and one of them wants to use theirs. Who's on first?

Since the TOC is already switched during procedure calls, that becomes a handy place to store the head of the resource chain. The linker could reserve space, say starting at zero. One word of that is assigned to the resource manager, say at offset 4. This word would be initialized to zero by the linker. The resource manager would store the application's "standard" head in a static variable of its own, as suggested above for the QuickDraw globals pointer. It would also be stored in the application's TOC. When asked to open a resource file it would look in the TOC of the caller and if the word is zero add the new file to the standard chain then store this new chain in the caller's TOC. If the TOC word is non-zero the new file gets added to that chain. LoadResource would use the chain from its caller's TOC.

Further complications, such as how to track and close all resource files at exit and what LoadResource should do if its caller's TOC does not have a resource chain, are left as exercises.

——— Open Issue ———

The example given of use by the Resource Manager is not a complete proposal. The calling conventions do make it possible to find the caller's TOC in certain cases. Calls within a shared library may not "do the right thing", depending on the compiler and linker used. This is discussed later.

——— Implementation Note ———

We can "fool" the IBM linker into "reserving" slots at the beginning of the TOC by providing a special module at the front of a link. This module would ensure that the reserved TOC entries get created at the right location. We would be depending on the implicit linker behavior for TOC layout. The Apple linker will always reserve space.

——— Open Issue ———

One of the interesting aspects of adopting the AIX runtime model is the opportunity to utilize AIX software easily under the Macintosh O/S. This may be rather difficult for arbitrary UNIX applications, but should be easier for standalone shared libraries. Such libraries won't have this reserved TOC space, but then they shouldn't be dealing with the Macintosh Toolbox either. We need to document what UNIX software can be used, how to use it, and what might go wrong.

We've been speaking of just a single TOC register. The architecture in fact only defines one dedicated register as a TOC pointer. With a 16 bit offset on the PowerPC this limits a TOC to 64KB, or 16K pointers. The extension of the architecture to use multiple TOC registers could obviously be made without conceptual violence. This was considered and rejected as unnecessary. The nature of the TOC itself, including code generation conventions, results in surprisingly modest demands on it. Other manual and tool-based approaches can reduce the risk of TOC overflow. Defining a second TOC register would remove that register from general use to the benefit of a very small number of "pig" programs.

The TOC provides addresses for the static data items used by a fragment and the routines in other fragments that are called. It does not provide a complete dictionary for its own fragment, nor is it a catenation of complete dictionaries for referenced fragments. Nor does it contain pointers to data and routines that are not actually used. Compiler conventions help, such as C's use of a single TOC pointer for all "static" variables in a compilation. It is rather difficult and unusual to actually construct a body of software that makes use of over 16,000 external variables and routines.

——— Rationale Note ———

For example, we have taken a pure C++ version of MacApp and built an AIX shared library from it to study TOC use and other issues. The shared library exports over 4500 symbols. We arbitrarily made every external variable and routine an export, although some should probably be kept internal to MacApp itself. The TOC for the MacApp shared library contains about 1050 pointers.

All Macintosh Toolbox services used by MacApp are treated as cross-fragment calls, each has a TOC pointer. There are around 300 of these. As an artifact of the MPW C headers and their conversion for test use on the PowerPC, references to low memory globals appears as pointer casts of literals, not as references to external data. But there are only about 25 low mems accessible through the MPW headers, so this is noise.

Since there are currently around 2000 to 2500 routines defined in the MPW headers and around 220 low memory globals in the IM X–Ref, a worst case scenario for a "full" PowerPC implementation would give MacApp a TOC with less than 4000 pointers.

The TOC for a client of MacApp would only contain what it used, an amount related to how much of MacApp was overridden. In a worst case of overriding everything and calling all system services directly, the client would have a TOC with around 7000 pointers plus its own static data, C library stuff, etc.

The size of the TOC could be manually managed by separating major subsystems into shared libraries of their own. This would require no source changes, only possible changes in linker scripts to name the additional libraries. It is quite likely that large pieces of software have large numbers of external references that can be segregated into subsystems, where the connections between the subsystems are fewer in number. This allows each subsystem to have a TOC dedicated to its own external needs and interconnections, without overcrowding from the external needs of others.

A linker could reduce the risk of TOC overflow by producing multiple TOCs in one fragment. All routines from one compilation would use a common TOC. Separate compilations could share a TOC as long as the TOC did not overflow. Since separate compilations are prepared for cross-fragment calls anyway, the linker can freely make them perform TOC switches within a fragment. This simple view only works for C though. More modular languages with hierarchical separate compilation might well require that subsidiary compilation units share the TOC of their top level ancestor.

——— Implementation Note ———

The PowerPC has 16-bit signed offsets in its memory reference instructions. To support a full 64KB TOC requires using the full range from –32,768 to 32,767. There are a variety of ways to do this, the choice being almost entirely up to the linker. A debugger might need to know what the linker did, but could accept any approach. We propose to first assign from zero to 32,767. Then if more space is needed it will be assigned downward from –1 to –32,768. The TOC is always a contiguous piece of storage. RTOC points to the beginning if the size is 32KB or less. RTOC points 32KB before the high address end if the size is greater than 32KB.

——— Rationale Note ———

One possibility for addressing a 64KB TOC is to use a "biased" TOC pointer, pointing 32KB beyond the actual start of the TOC. Items in the TOC would have offsets starting at –32,768 and ranging steadily up through zero to 32,767. This has the advantage of being simple to describe and easy to layout. Another possibility is to first assign from zero to 32,767. Then if more space is needed start assigning downward from –1 to –32,768. This meets naive expectations in the common case of a TOC smaller than 32KB. It also preserves compatibility with IBM in that case. A third possibility is to place the TOC pointer in the middle, using offsets from –n/2 to n/2. This doesn't seem to have any unique advantages.

All three approaches have a contiguous, minimally sized block of storage for the TOC. Since tests so far indicate that a TOC larger than 32KB is unlikely, the second approach provides a simple rule in virtually all cases. Being compatible with IBM also allows us to use the previously mentioned trick to provide reserved slots in the TOC at fixed offsets. The biased pointer approach was previously advocated, then dropped as the above reasoning evolved.

——— Management Note ———

IBM's tools presently use only non-negative offsets, limiting the TOC to 32KB. They have talked about having their compilers place small static objects directly in the TOC and having their linker extend the limit beyond 32KB. We should track their plans and try to maintain consistency, perhaps through the PowerOpen group.

Enough TOC, Just Do It!

——— Reality Check ———

There are a lot of things that may happen with fragments and the TOC, but in point of fact today's linker creates just one code and data section for each fragment. The linker also creates just one TOC per fragment buried within the fragment's data section. Under UNIX the .text section corresponds to our code section and the .data and .bss sections together correspond to our data section. Compilers do not presently put any data directly in the TOC. IBM's linker only supports a 32KB TOC, using just non-negative offsets. Apple's linker will support a 64KB TOC as described earlier.

——— Unreality Check ———

Reality not withstanding, programmers should not assume that there is just one code and data section per fragment, nor just one TOC in that one data section. Assumptions should not be made about the offsets used for TOC entries. This is especially true for debugger writers. The TOC is something that allows a routine to find its way in the world. Static variables have offsets within data sections where they are allocated. The offset of any particular TOC pointer to a static variable is relevant only to code using that TOC to access the variable. A debugger could go directly to the location, not using any particular TOC. Similar arguments hold for external routines.

VI. Subprogram Invocation

This section presents some general calling conventions for any platform and specific code sequences for the PowerPC. It only covers calls and returns, not parameter passing. The framework for this section is the invocation of a parameterless procedure containing nothing but a return instruction. While any compiler is free to use its own conventions internally, these are the "Apple standard" conventions that will be followed by Apple language products. They are largely the conventions developed by IBM for AIX on the RS/6000. There are some minor preferences that differ. These are pointed out below.

There are three flavors of calls to consider, local, cross-TOC, and pointer based. Local calls are known at compile-time to not require a change of the TOC register. Typically these are calls within a source file. Cross-TOC calls may require a change of the TOC register, whether this is actually true is not determined until link time. Typically these are calls outside of a source file. Code generation for cross-TOC calls entails only minor overhead on the caller's side, additional code may be created by the linker. Pointer based calls are those through procedure variables. They always "switch" the TOC register, even though the new value may be the same as the old. The called routine is always ignorant of the nature of the call. The TOC register is presumed to be properly set on entry to any routine.

Local Calls

Local calls use PC relative branches, straight to the destination routine. As should be expected they involve no additional overhead. Calls to C "static" routines should always be local.

Calls to "extern" routines in the same source file are not so obvious. At first glance they won't involve a TOC switch if the linker produces just one TOC per fragment, or at most one TOC per compilation unit. However, a "patching" mechanism has been designed for PowerPC shared libraries that provides analogous capabilities to the 680x0 trap patching mechanism. The PowerPC patching mechanism depends on aspects of the cross-TOC call implementation. It is desirable for C compilers to provide an option to make all calls to "extern" routines be compiled like cross-TOC calls. The linker should have an option to make calls to exported routines actually utilize the cross-TOC mechanism so that patching will work. Were this not done calls within the library to patched routines would bypass the patches.

——— Implementation Note ———

The PowerPC unconditional branch has a 24 bit signed instruction offset, allowing simple code generation for a code section of up to 32MB. Unlike the 680x0, the PowerPC is intelligent enough to know that instructions are in 32-bit chunks and that using byte offsets for instructions is silly and wasteful. The 16-bit signed byte offsets on the 680x0 are not an insurmountable barrier but do require additional linker smarts to create branch islands and the like.

——— Implementation Note ———

At the time of this writing the compilers from IBM do not supply the suggested option. Later versions will. The IBM linker will not ever supply the corresponding option, the Apple linker will.

Routine Descriptors

An earlier note anticipated the need to understand routine descriptors. The story may now be told.

Each externally visible routine has a descriptor associated with it. The descriptor is created by the compiler as a normal piece of static data in the same unit as the code for the routine. The descriptor does not, repeat, does not live in the TOC. There is a one–to–one association of descriptors and routines. The descriptor is always in the same fragment as the code of the routine that it describes. At the language level a pointer to a routine is actually a pointer to its descriptor, not the code entry point.

The descriptor may be two or three words long at the discretion of the compiler. The first word is the entry point of the associated routine. The second word is the TOC address for that routine. These two words are another example of non-TOC static data that is relocated by the runtime loader. The third word, if present, is an undefined "environment" word.

——— Implementation Note ———

The third word of the descriptor is not used by typical C compilers, although IBM's compilers do allocate it. A language such as Pascal with nested routines might use the environment word as the static link. Taking the address of a nested routine results in the creation of a descriptor whose third word is filled in with the latest stack frame address for the routine's lexical parent. Other languages may define their own usage.

The descriptor contains the code address for the routine and the address of the TOC for the routine. As you might guess, the descriptor contents are relocated as part of the runtime loading process. This is one of the examples of non-TOC relocation mentioned earlier. The TOC pointer involved in cross-fragment calls points to the descriptor. Each caller has a pointer to the descriptor in its own TOC, just like references to any external variable. Glue code involved in cross-fragment calls uses this TOC pointer. It saves the current value of RTOC, loads the new RTOC value from the descriptor and branches to the code address in the descriptor. The original RTOC value is restored on return. This will all be detailed in a later section on calling conventions.

——— Caveat Coder ———

The Mixed Mode mechanism also has a notion of descriptor for routines, with an uncomfortably similar name. We need better terms.

Cross-TOC Calls

Cross-TOC calls are generated by compilers as a PC relative branch followed by an instruction to restore the TOC register. If the linker determines that the call is in fact local it will replace the TOC reload instruction with a NOP instruction. Otherwise the linker will create a small, customized piece of glue code in the code section of the call. The compiler generated branch is directed to this glue. There is one piece of glue in each fragment for each distinct destination routine.

——— **Implementation Note** ———

In actuality linkers must be prepared to recognize the TOC reload or at least three possible NOP instructions and do the right thing. The POWER (hardware) architecture specified one preferred NOP, early versions of the PowerPC architecture a second and current versions of the PowerPC architecture a third. There is no single NOP instruction. One of many "ineffectual" instructions is defined as preferred by CPUs.

The glue saves the current TOC register contents, loads the new value, and jumps to the actual destination. The caller reserves a standard slot in its stack frame to save the TOC register. The return is made to the original caller, specifically to the TOC reload instruction. Any epilogue for parameter cleanup or function results is performed after this.

The linker created glue uses the first two words of the destination routine's descriptor to obtain the destination address and new TOC value. The third word of the descriptor, if any, is ignored by this glue. The glue accesses the descriptor just like any static variable, via a pointer in the TOC of the calling fragment. The glue is a stub inside the calling fragment. The linker will create this TOC pointer if necessary.

——— **Implementation Note** ———

These cross-TOC calls are optimized on the presumption that they are to top level routines and that common procedural languages dominate which do not need the environment word for top level routines. If the destination never needs the environment word it is silly to pass it. (Please read the pointer based call section before flaming.)

On the PowerPC the glue to call a routine named foo would be something like:

```
   bl     foo_glue                     # Call the cross-TOC glue
   lwz    RTOC, TOC_save_offset(SP)    # Restore the caller's TOC pointer
   ...

foo_glue:
   lwz    R12, descr_of_foo(RTOC)      # Get pointer to foo's descriptor
   stw    RTOC, TOC_save_offset(SP)    # Save the caller's TOC pointer
   lwz    R0, 0(R12)                   # Get foo's entry point
   lwz    RTOC, 4(R12)                 # Load foo's TOC pointer
   mtctr  R0                           # Move entry point to count register
   bctr                                #  and jump to foo
```

——— Explanatory Note ———

The compiler generates the call as a call to "foo". The linker figures out that this is a cross-TOC call, adds the glue, and directs the call there. Custom glue for each destination is necessary because the glue has embedded in it the offset of the TOC pointer to the descriptor of the destination routine.

——— Implementation Note ———

This is the code currently generated by IBM's compilers and linker. We could do slightly better with our own compilers and linker at the risk of not being able to link Apple compiled code with an IBM linker. The compiler could emit a TOC save instruction as part of a routine's prologue, tagged by a special form of relocation. If the linker found any cross-TOC calls in the routine it would leave the reload alone, otherwise it would replace it with a NOP. The glue could then omit the TOC save instruction. This would reduce the size of the glue but possibly not its speed since the TOC save sits in the delay slot between the load of the pointer to the descriptor and the load of the entry point from the descriptor.

——— Management Note ———

Upcoming versions of AIX are reported to drop use of the descriptor for cross-TOC calls. They might place the code address and new TOC address directly in the caller's TOC. This saves one instruction in the glue. However the proposed "patching" strategy for Apple relies on the use of the descriptors. Internally we can simply stick to use of the current IBM linker. We should address this issue in the PowerOpen standards since it affects binary compatibility. Libraries imported with the new IBM model would operate OK except that patches applied to them would not always be called. (In fact this has not yet happened as of AIX 3.2.)

Pointer Based Calls

As mentioned earlier, taking the address of a routine or passing a routine as a parameter uses the address of the routine descriptor since the eventual call may be cross-TOC. This is the address of the descriptor, not the descriptor itself. That is, a routine pointer is still a 32-bit pointer albeit to the descriptor instead of the code.

The instruction sequence in this case is very similar to that for the cross-TOC call, with just two significant differences. The compiler generates three instructions to load the descriptor pointer into a scratch register, to call a standard glue routine, and to reload the caller's TOC. This standard glue routine operates like the cross-TOC glue, in addition it loads the environment word into a standard register.

——— Implementation Note ———

Passing the environment word allows nested routines in languages like Pascal to be properly passed as pointers and called from anywhere, even from other languages. This also provides flexibility to other language models that wish to always use the environment pointer. They may cast their external API in terms of routine pointers instead of direct routine names. This allows them to still interoperate cleanly with other languages.

The code for the PowerPC to call through a routine pointer might be something like:

```
    lwz    R11, address_of_descriptor
    bl     ptr_glue
    lwz    RTOC, TOC_save_offset(SP)      # Restore the caller's TOC pointer
    ...

ptr_glue:
    lwz    R0, 0(R11)                     # Get the entry point
    stw    RTOC, TOC_save_offset(SP)      # Save the caller's TOC pointer
    mtctr  R0                             # Move entry point to count register
    lwz    RTOC, 4(R11)                   # Load the new TOC pointer
    lwz    R11, 8(R11)                    # Load the environment pointer
    bctr                                  # Jump through the count register
```

——— Implementation Note ———

The optimization of the TOC save mentioned above should obviously be extended to include pointer based calls.

——— Open Issue ———

The astute reader will notice that the cross-TOC example loaded the descriptor pointer into R12 and the pointer based example uses R11. These are the actual code sequences currently used by IBM's tools. There may be advantages to low level bits of software like debuggers or the patching mechanism if common conventions were used. The flexibility is limited since the caller of ptr_glue passes the pointer in a non-standard register. This logic is coded into compilers. Similarly coded into compilers is the assumption that the environment word is passed in R11. The proposed approach is to define another version of ptr_glue with a different name that takes the descriptor address in R12. Compilers could evolve to use that version over time. This allows new code to call existing binaries properly and vice versa.

VII. Stack Frames

Stack Frame Organization

The stack layout is conventional enough to appear familiar, yet different enough to clobber the unwary. The most significant difference is the use of just one stack pointer instead of two. Another is that a bounded amount of memory beyond the end of the stack may be used. A third difference is that stack frame boundaries are blurred, with portions of the caller's frame explicitly used by the callee.

——— Explanatory Note ———

The almost universal convention for grow-down stacks is to have one register point to the low address end of the stack, the stack top or stack pointer, and another point to the high address end of the stack frame, the frame pointer. Parameters are addressed with positive offsets from the frame pointer and local variables with negative offsets from the frame pointer. The stack top floats during expression evaluation, parameter passing, etc., as values are pushed and popped.

This model uses a grow-down stack with a single pointer at the low address end. The stack pointer does not generally move after the frame is created. This ignores the case of dynamically sized local variables, which is discussed later. The definition of stack frame here is the section of memory between the caller's stack pointer and the callee's stack pointer.

——— Rationale Note ———

This model is optimized for the predominant case of routines with fixed sized frames whose size is known at compile time and for frameless leaf routines. C does not allow the declaration of variable sized locals. In those languages that do, most routines still have fixed size frames.

Of course the protocols defined here are loose standards in that any language can use almost any approach it desires for execution within its own domain. Adherence to these standards will greatly simplify the development of multilingual tools such as debuggers, applications with components written in different languages, callback routines written in arbitrary languages, and software substitutions (analogous to trap patching on the 680x0).

Figure 4 shows the general layout of the stack. The stack is drawn using IBM's convention of low addresses on the top, counter to typical 680x0 usage. This makes it easier to visualize some of the sections as structures with positive offsets. Taking that view makes it easier to understand and accept some of the unconventional aspects of this model.

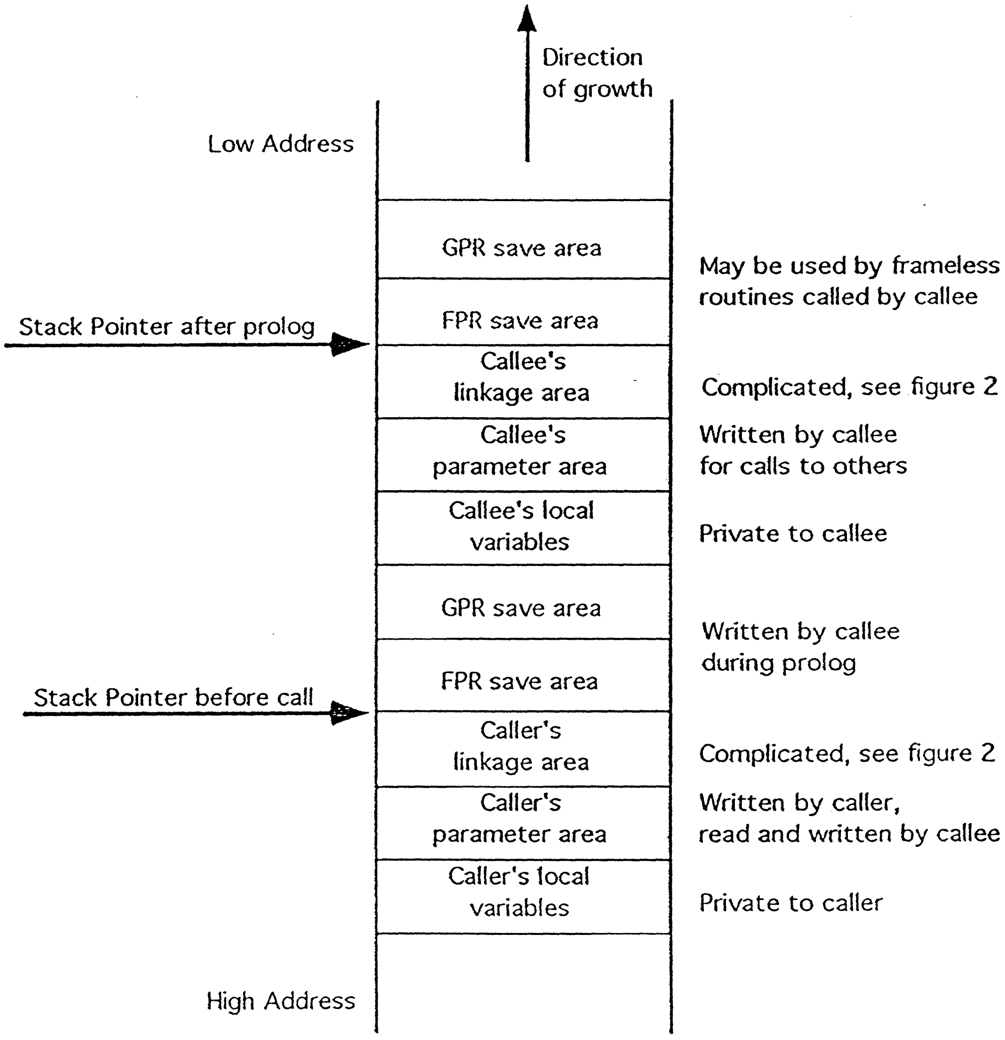


Figure 4. Stack layout

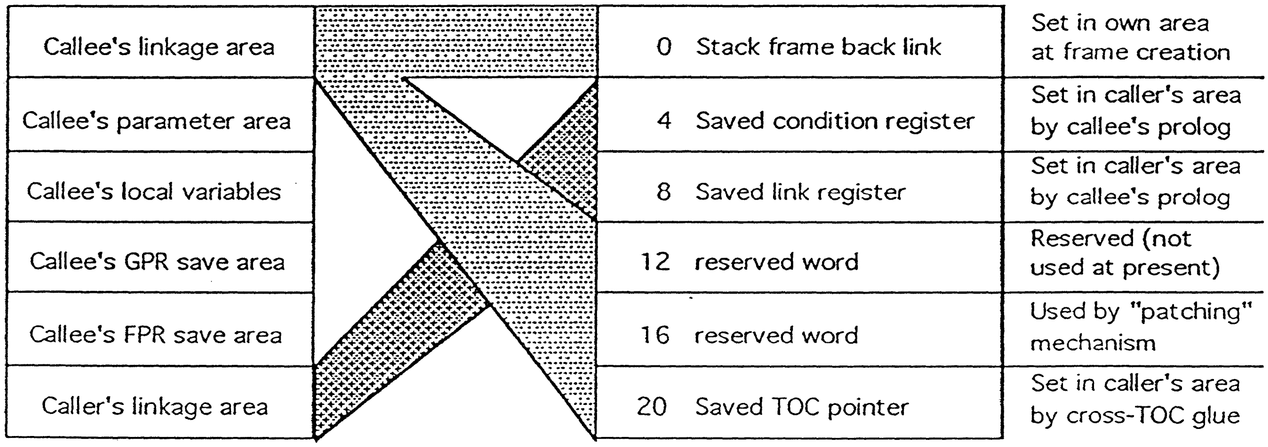
The callee's frame is the memory between the two stack pointer values, although the callee also uses portions of the caller's linkage and parameter areas. Working from the before call stack pointer up, the FPR and GPR save areas are used by the callee to save those non-volatile registers that it uses. The local variables are obvious. The parameter area is used to store parameters passed to called routines. The linkage area contains a six word structure described later. Finally the FPR and GPR save areas beyond the new stack pointer are used by frameless leaf routines called by the callee. This expands the powers of frameless routines by allowing them to use non-volatile registers.

The caller's parameter area is also known as the callee's argument area. When the callee is running the parameters passed to it are in the callee's argument area, the same as the caller's parameter area. The callee's parameter area is used to pass parameters to other routines called by the callee.

——— **Implementation Note** ———

The save area on the low address side of the stack pointer is used only as needed by frameless leaf routines. They only save as many of the non-volatile registers as they use. However any software operating asynchronously on the stack must account for the largest possible save area. The stack pointer must be decremented by that amount before anything is placed on the stack by the asynchronous operation. This amount is 224 bytes, which includes alignment to a quadword boundary.

Ron Hochsprung has dubbed these save areas the "Red Zone".



0	Stack frame back link	Set in own area at frame creation
4	Saved condition register	Set in caller's area by callee's prolog
8	Saved link register	Set in caller's area by callee's prolog
12	reserved word	Reserved (not used at present)
16	reserved word	Used by "patching" mechanism
20	Saved TOC pointer	Set in caller's area by cross-TOC glue

Figure 5. Details of stack frame linkage area

Figure 5 shows the structure and use of the stack frame linkage area. Portions of the linkage area are used by the caller and other portions by the callee. Put another way, the callee uses portions of the callers linkage area and portions of its own linkage area, with the unused portions of the callee's own linkage area used by other routines that it calls.

The stack frame back link points to the immediately preceding frame, i.e. the contents of SP before the call. It is set by the callee in its own linkage area when creating the frame. The saved TOC pointer is the caller's RTOC contents. It is saved by the glue for cross-TOC calls in the caller's linkage area before jumping to the callee and restored by the caller after return. The saved condition and link register slots are set by the callee in the caller's linkage area if the callee modifies those registers. The use of the reserved "patching" word is described in a different document.

——— Explanatory Note ———

The diagram may look funny because the saved TOC slot is connected to the callee's linkage area. The drawing is from the view of callee. The saved TOC slot in the callee's linkage area holds the callee's TOC pointer when it calls other routines. A routine reloads its RTOC from the save slot in its own frame after a cross–TOC call.

——— Open Issue ———

IBM documentation claims that the word at offset 12 is used by the C library routines setjmp and longjmp. Disassembly of their code and tracing with a debugger did not show this use. We're assuming that the documentation is out of date. The use of these fields should be coordinated with IBM to enhance interoperability of application and shared library binaries between Macintosh and PowerOpen. This is best approached through the PowerOpen standards but the mechanism for influencing them is not clear.

On the PowerPC the stack should be kept quadword aligned. It must be at least word aligned for the load–multiple-word and store–multiple-word instructions to operate at all. For performance the high address end of the GPR save area should be quadword aligned. The suggested approach is to save only the necessary FPRs, possibly leave an alignment gap between the GPR and FPR save areas, save only the necessary number of GPRs, and possibly leave another alignment gap between the local variables and the GPR save area.

——— Explanatory Note ———

The load–multiple–word and store–multiple–word instructions take a starting register number and save from that register through R31. The PowerPC hardware architecture states a preference for the contents of R31 to be the last word in a quadword. This is true for both 32-bit and 64-bit CPUs, and for both the word and doubleword loads and stores. On 64-bit CPUs, the load–multiple–doubleword and store–multiple–doubleword instructions will require doubleword alignment. Using just word alignment saves little space, slows everybody else down, and will break 64-bit software. Don't do it.

——— Explanatory Note ———

Computing the size of this second gap is a little tricky because sizes of the GPR save area, local variables, and parameter area must all be considered. Also, the linkage area has six words, throwing simple expectations off by two words.

——— Explanatory Note ———

The suggested approach should slightly improve performance in two ways. First the minimum number of non-volatile registers are saved and restored. Putting the second gap between the locals and GPRs keeps the parameters and locals together, increasing the chance of keeping dead space (the gap) out of the data cache. The gap cannot go between the parameter and the linkage areas, the parameters must start at offset 24.

Some Apple operating systems may provide guard space at the low address end of a stack to support overflow checking. Where this is the case the guard space will be at least 4K bytes long. Safe checking for larger allocations requires probes in 4K byte increments or use of a checking service.

——— Open Issue ———

The 4KB size was arbitrarily chosen as a likely virtual memory page size. On the PowerPC frames up to 32KB can be created in a single instruction. Larger frames require first computing the frame size in a register. Perhaps 32KB would be a better buffer size, requiring a check only when extra code is being generated anyway.

The standard stack frame model does not specify a mechanism for up level references in languages that support nested routines. The choice of static links or displays is controversial and very dependent on usage patterns, optimization technology, available registers, etc. This is left up to individual compiler implementors.

Using the Stack

The usage of the stack is best understood in terms of three phases of subprogram calls:

1. Processing parameters before the call instruction,
2. Glue and prologue before frame creation, and
3. Normal execution after frame creation.

First, the caller places each parameter in the caller's parameter area. The layout of this area is defined later. Since all calls use this one parameter area, temporary storage in non-volatile registers or the local variable area must be used when parameter lists themselves contain calls with parameters. After the parameters are prepared the call is made.

——— Explanatory Note ———

The discussion of parameter passing is slightly obscured by the fact that parameters are usually passed in registers. We'll ignore this for now.

——— Explanatory Note ———

The problem of parameter processing for nested calls simply does not occur for models that push parameters on the stack. For example in "foo(a, bar(i, j, k), c)" , the values of a and c cannot be put in the parameter area until after the call of bar since a and i use the same slot as do c and k.

If this is a cross-TOC call, control goes next to the glue stub in the current code section. As seen before, the stub will save the current value of RTOC in the last word of the caller's stack linkage area, load the new RTOC value, and branch to the actual callee. If this is not a cross-TOC call control goes directly to the callee.

The callee's prologue will create the frame, store the back link, save the link register if any non-leaf routines are called, save the entire condition register if any of its non-volatile fields are used, and save any non-volatile GPRs and FPRs that are used. The frame creation and store of the back link are actually done with a single store–word–with–update instruction.

——— Implementation Note ———

As suggested in an earlier note, we may include saving the TOC value in the prologue.

——— Implementation Note ———

The prologue may save the registers either before or after creating the frame. The link register and condition register are saved in the caller's linkage area, which is already available. The GPRs and FPRs may be saved before frame creation by taking advantage of the "Red Zone"™. Allocation of the frame will then magically neutralize the red menace.

——— Rationale Note ———

One could imagine "optimizing" the code to save registers only when first needed, especially if conditional paths had quite different needs. Saving the registers within the prologue can greatly simplify any facilities that need to unwind stack frames and do so by examining code. Examples include the proposed C++ exception mechanism and display of register based variables during debugging. On high–end CPUs the store–multiple instruction is likely to receive particular hardware attention.

These protocols allow leaf routines (those that make no calls) to do non-trivial work without creating a stack frame. This can give a considerable performance improvement, especially with modern trends towards large numbers of very small routines. If parameters, local variables, results, and the return address all stay in registers the only memory references made by leaf routines will be for code.

Possible PowerPC prologue code for a routine might be something like:

```
.foo:
   mflr  R0                         # Extract the return address
   mfcr  R12                        # Extract the condition register
   bl    $SaveFPR25                 # Save FPR25-FPR31
   stmw  R18, -120(SP)              # Save GPR18-GPR31
   stw   R0, 8(SP)                  # Save the return address
   stw   R12, 4(SP)                 # Save the condition register
   stwu  SP, -208(SP)               # Allocate the new frame
```

——— Implementation Note ———

The $SaveFPRn routines are hypothetical standard runtime routines that save from FPRn through FPR31 into the "Red Zone"™. There is no store multiple instruction on the PowerPC for the floating point registers. The GPR save offset includes alignment padding, being computed as "-56-8-56" for "-FPRSize-alignment-GPRSize". The stack allocation includes 64 bytes for the parameter area, local variables, and alignment. The division of this space is left as an exercise to the reader.

——— Open Issue ———

The $SaveFPRn routines cannot be in a shared library. The simplest approach is to get them from a standard linker library, i.e. have them copied into every fragment that needs them. Since all PowerPC branches can be relative or absolute, the unconditional branch–and–link can go straight to the low or high 32MB of memory. AIX takes advantage of this for common utility routines, perhaps we should too.

This needs to be verified with a real example. AIX documentation and reality are not always in synch. If true it represents a serious compatibility problem in using AIX binaries under the Macintosh O/S.

——— Implementation Note ———

Use of the store-with-update instruction to create the frame is important to guarantee that the stack linkage is always valid, making life a bit nicer for debuggers.

——— Implementation Note ———

On the PowerPC the largest stack frame that may be allocated with the one instruction shown above is 32KB. Larger frames are created by first computing the size in a register then using the indexed form of the store–word–with–update instruction. R0 and R12 are available, as is R11 if not used as for the environment word. R13-R31 are available if saved first.

Finally the callee will access its parameters directly from the caller's parameter area (a.k.a. callee's argument area). Note that this is done without indirection through the back link since the callee knows its own frame size and the parameters always start 24 bytes beyond the end of its frame. The callee's parameter area is used to pass parameters to routines that it calls.

The epilogue code is very analogous to the prologue code. One performance hint is to deallocate the frame by incrementing the stack pointer instead of loading the back link. This saves a memory reference.

Dynamically Sized Locals

C does not directly support dynamically sized local variables, although the alloca service does allow their manual creation. Other languages support them directly. Dynamically sized locals are generally allocated "beyond" the fixed size portion of the stack frame and accessed through a pointer in the fixed size portion. They require separate notions of a frame pointer for addressing the fixed size portion of the frame and a stack pointer to denote the end of allocation for the stack.

Because called routines utilize the caller's linkage and parameter areas, these portions of the standard stack frame must always be located at the top of the stack. A dynamic allocation proceeds as follows:

- Create a frame pointer if this is the first dynamic allocation.
- Decrement the stack pointer by the size of the allocation, maintaining the back link and copying the saved RTOC value (if any).
- Set the new object's address to SP+24+Param_Size, i.e. between the new linkage/parameter areas and the fixed size local variables.
- Address the fixed size portion of the frame via the frame pointer from now on.

——— Implementation Note ———

The size of the allocation and the new objects address may both need padding to maintain quadword alignment of the stack and to properly align the new object itself.

——— Implementation Note ———

Since the use of the frame pointer is entirely local to a routine there are no defined rules in the architecture. Common practice would reduce debugging confusion though. We suggest that R31 be used as the frame pointer, that it be captured in the prologue just after frame creation, and that its value be the initial stack pointer value. This last point keeps variable offsets from the frame pointer "consistent" with those from the stack pointer, again to reduce debugging confusion.

When doing a dynamic allocation, the "Red Zone"™ is known to be empty, it is for frameless leaf routines. Except for the stack back link and saved RTOC slot the linkage area is also empty. The link register and condition register save slots are for called routines. The use of the reserved words is constrained by definition to not require copying them.

——— Implementation Note ———

The cross-TOC glue shown earlier saves RTOC on each call, so the save slot would not need to be copied. If the suggested optimization to save RTOC just once in the prologue were implemented, then the RTOC save slot would need to be copied.

The copying of the parameter area depends on programming language semantics and the details of a particular allocation, specifically whether the allocation occurs in the midst of a processing a parameter list. A language such as C which performs dynamic stack allocations through a library routine (alloca) is one example. Others include languages with dynamically sized declarations and inline routine expansion or blocks within expressions.

Figure 6 shows shows the stack after two dynamic allocations. Note the the frame pointer points where the stack pointer did before the first dynamic allocation.

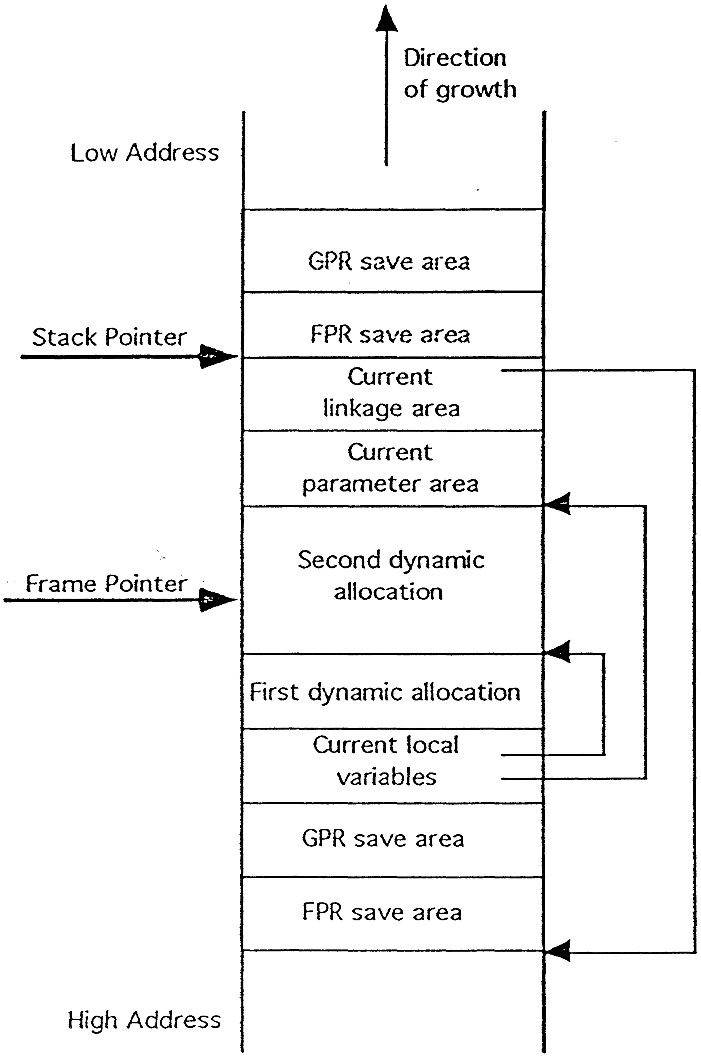


Figure 6. Dynamically sized local variables

VIII. Parameter Passing

The parameter passing protocols utilize registers for the transmission of most arguments with a shadow storage area in the caller's frame. It is best viewed by first considering the caller's parameter area as a structure containing all of the parameter values, then using registers as an optimization. There is just one parameter area in each frame, it must be as large as the largest parameter list used by calls from that routine. The internal use of the parameter area, i.e. the mapping of parameters to parameter area offsets, is of course determined for each call.

The parameters are laid out in the parameter area in textual order with the left most parameter at the lowest offset. Each parameter starts at a word boundary regardless of size (e.g. characters occupy a word and double floats are not necessarily on a double word boundary).

—— Implementation Note ——

Ideally double floats should be on a doubleword boundary. However this is the current AIX standard and is retained for compatibility.. As will be shown later, floating point parameters, single and double precision, will almost always be passed in registers. The effect of odd word alignment will hopefully be minimal.

Of course parameters passed by reference use one word for the address. Parameters passed by copy may be either input or output parameters. Neither C nor Pascal support output parameters by copy but other languages do. Byte and halfword scalars (things with integer-like values) that are passed by copy are sign extended to a word. Unsigned bytes and halfwords are of course zero extended. Composite types that are passed by copy use as many words as necessary with extra space at the high address end being undefined. Floats and double floats take one and two words respectively.

—— Implementation Note ——

The C notion of extending shorter values to words is standardized for all languages to simplify interlanguage calls. It is worth noting that the PowerPC can load and zero or sign extend half words in one instruction but can only load and zero extend bytes. All four cases can be handled in one instruction for values already in a register.

—— Implementation Note ——

There is a feature in the IBM C compiler where it uses a sequence of word loads for composites, fetching up to 3 extra bytes beyond the needed storage. This would cause problems if say a six byte array happens to end exactly at an addressing boundary. You'll get an access error for the extra two bytes that you don't even want. The IBM linker and AIX avoid this by always padding data sections with an extra word. A doubleword pad should probably be used in anticipation of using doubleword loads on 64-bit CPU implementations.

The method of passing is language dependent. C has only input parameters, passes all scalars and structures by value, and all arrays by reference. Pascal passes scalar input parameters by copy, record and array input parameters by reference with callee copy, and all VAR parameters by reference. Other languages may have different rules, for example FORTRAN passes all parameters by reference and Ada passes all scalars by copy whether input or output. It is up to each compiler vendor to define how to call subprograms written in other languages and to document the protocols used for internal calls.

This does not imply arbitrary deviation from these standards but documentation of the language implementation and suggested mappings to other languages. For example, in-out scalars could not be passed between C and Ada unless the Ada compiler allowed selection of reference passing to the C routine. In that case the C routine would pretend that it got a pointer, or a ref for C++.

Figure 7 shows the layout of the parameter area for the following C procedure. The offsets are those from the stack pointer as the caller prepares the parameters.

```
typedef struct rec {short s1, s2, s3;} rec;
void foo (char c1, char *c2, short s1,
          long a[], rec r1, rec *r2);
```

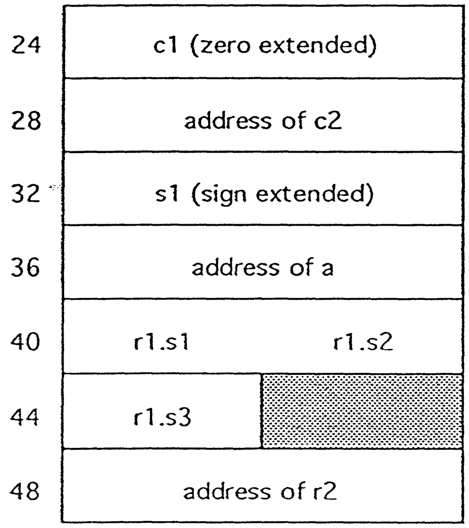


Figure 7. Parameter area example

A number of registers on the PowerPC are utilized for parameter passing in order to avoid memory references to the parameter area. The rules governing this are not complicated but are easy to misunderstand.

The first rule is that the first eight words of the parameter area are mapped to the general purpose registers R3 through R10. Except for floating point parameters, these eight words are placed in the corresponding register on a word–for–word basis. Scalars and pointers occupy one register each, composites occupy as many consecutive registers as needed. Note that composites are mapped to the registers as a memory image, not as components. An array of six bytes uses two registers, all of the first and the top half of the second. The lower half of the second is undefined, as is the lower half of the corresponding parameter area word.

The next rule is that the first 13 floating point parameters are passed in the floating point registers FPR1 through FPR13. This only applies to "free" float parameters, floats embedded in composites do not count. If the floating point parameter appears within the first eight words the corresponding general register or register pair is not used. The general register is not used later, it is simply skipped. The floating point registers are used for consecutive floating point values, they are not mapped to fixed slots in the parameter area like the general registers.

The final rule is that all values not in registers are in fact stored in the parameter area. Note that a multi–word composite parameter may be partially in the GPRs and partially in the parameter area if it starts in the eighth word or earlier and ends after the eighth word.

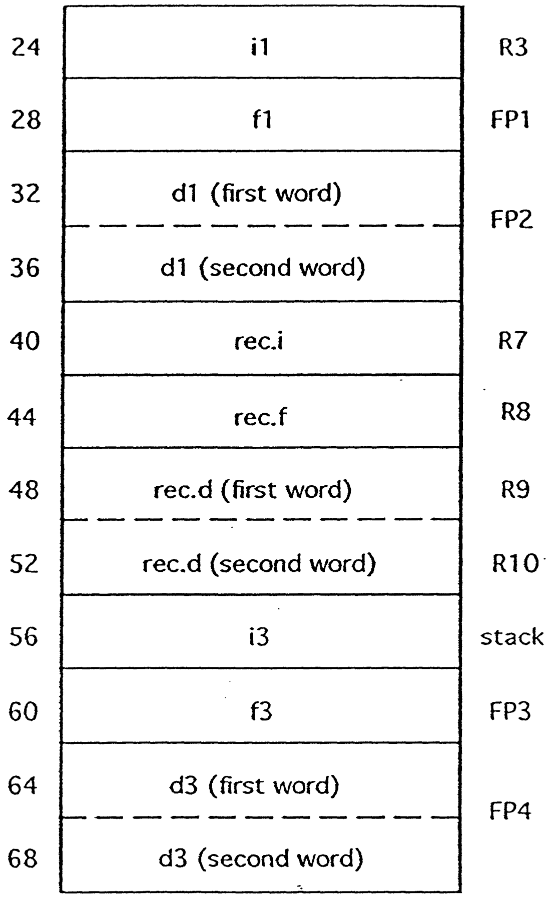
Figure 8 shows the parameter layout and register usage for the following C procedure:

```
typedef struct rec {int i; float f; double d;} rec;
void foo (int i1, float f1, double d1, rec r,
          int i3, float f3, double d3);
```

There are three things to take not of. First, R4, R5, and R6 are not used. Second, the floating point fields of the structure are not "pulled out" into the floating point registers. Third, the floating point registers are used contiguously although the float parameters are not contiguous.

——— Picture Fix ———

Tweak the example to show the structure partially in registers and partially on the stack.



24	i1	R3
28	f1	FP1
32	d1 (first word)	FP2
36	d1 (second word)	
40	rec.i	R7
44	rec.f	R8
48	rec.d (first word)	R9
52	rec.d (second word)	R10
56	i3	stack
60	f3	FP3
64	d3 (first word)	FP4
68	d3 (second word)	

Figure 8. Register mapping example

If a parameter value is passed in a register the corresponding space in the parameter area must still be available, but is not set by the caller. The callee may use the caller's parameter area to spill the parameter registers if desired. Parameter registers not used are considered scratch by the callee. The caller must store all parameter registers before calling other routines, even if those routines have few parameters, since they may in turn call routines with several parameters.

——— Implementation Note ———————————————————————

As an architectural rule the parameter area need be no bigger than the largest parameter list. C compilers have to modify this in three special cases. In the first case, routines with a variable number of parameters must be called with a parameter area no less than eight words long. These routines will store R3 through R10 on entry then access the parameters through a pointer into the caller's parameter area.

Routines that do not have prototypes must be assumed to have a variable number of parameters. In the second case, routines with floating point values in the ellipsis (...) portion of a variable length parameter list must pass those floating point values as non-floats, i.e. either in the GPRs or in the parameter area. These routines won't know they are getting floating point values for those parameters. Finally, C routines without prototypes must have floating point values passed both in the FPRs and GPRs/stack. Without a prototype the caller cannot tell if the callee expects a float or has a variable length parameter list.

The final rule applies to C calls when a prototype is not around for the callee. In this case floats are treated like both floats and non-floats. That is the first eight words are passed in the general registers regardless of whether or not they are floats. The first 13 floats are still passed in the floating registers. This is done because the callee could be expecting things either way. If the callee definition specifies floats then it will be looking in the floating registers. If it is a routine like printf that accepts anything and interprets it internally it will be looking in the general registers.

—— Implementation Note ——

IBM standardizes on a minimum 8 word parameter area

Function Results

Functions with scalar results return them in R3. Single and double precision floating point results are returned in FPR1, long double results are returned in FPR1 and FPR2. Composite results whose size is known by the caller are returned in storage allocated by the caller. The address of this storage is passed in as an implicit left most parameter, i.e. in R3.

—— Open Issue ——

The handling of function results may need more investigation. This covers the capabilities of just C and Pascal. At least FORTRAN should be covered in addition.

—— Open Issue ——

Some IBM documentation mentions all of R3-R10 and FPR1-FPR13 as function result registers, although it is not clear where extra registers are used. The only common known case is FORTRAN's COMPLEX type, returned in FPR1 and FPR2. LISP has also been mentioned as a possible user of more registers.

IX. PowerPC Register Conventions

Figures 9, 10, 11, and 12 summarize the general, floating point, condition register, and special purpose register usage conventions for the PowerPC. The condition register conventions are shown for each 4-bit field. A non–volatile register is guaranteed to have the same value upon return as before the call.

GPR	Volatile?	Usage
0	Yes	Scratch, used in glue and prologs
1	No	Stack pointer (SP)
2	No	TOC pointer (RTOC)
3-10	Yes	First 8 parameter words. R3 is used for non-float results.
11-12	Yes	Scratch, used in glue and prologs
13-31	No	Non-volatile local storage

Figure 9. General purpose register conventions

FPR	Volatile?	Usage
0	Yes	Scratch
1-13	Yes	First 13 floating point parameters. FPR1 & FPR 2 used for float results.
14-31	No	Non-volatile local storage

Figure 10. Floating point register conventions

CR	Volatile?	Usage
0-1	Yes	Scratch, set by fixed (0) and float (1) operations via record bit (Rc)
2-5	No	Non-volatile local storage
6-7	Yes	Scratch

Figure 11. Condition register conventions

SPR	Volatile?	Usage
LR	Yes	Procedure call and return
CTR	Yes	Loops, computed branches
PMR	No	Major CPU modes
XER	Yes	Fixed point status
FPSCR	Yes	Floating point status and control

Figure 12. Special purpose register conventions

X. Shared Libraries

This section discusses the inner workings of shared libraries. It also covers a couple of general issues that apply equally to any fragment. The discussion here is in more depth than appropriate for the earlier discussion on fragments.

There are actually two shared library systems to be aware of on the PowerPC Macintosh. The system described in this paper is an intrinsic part of the runtime architecture. It may never be available on 680x0 machines. It provides very transparent, easy to use facilities for traditional procedural languages. All use of the term "shared library" in this paper is to this intrinsic system.

The intrinsic shared library system is managed by the PowerPC Code Fragment Manager (CFM) in conjunction with the PowerPC Code Fragment Loader (CFL). The APIs for these are found elsewhere.

The second shared library system is focused specifically at sharing C++ classes. It will exist on both 680x0 and PowerPC machines. On the PowerPC it will be layered on top of the intrinsic system. This second system is documented elsewhere, and not discussed further here.

—— Implementation Note ——

The term "build time" is used frequently in this section. It refers to "around link time", i.e. after compilation and before execution, either during or after linking. Some of the operations discussed are not handled by IBM's linker and will be handled initially by post processing tools. These may or may not be rolled into Apple's linker.

Overview

As described earlier, the TOC addressing model provides independence among execution units. They can appear anywhere in memory relative to one another and only a comparatively small number of addresses in the TOC and other static data need to be relocated at load time. This model supports shared libraries with dynamic binding at low cost. Each concurrent use of a shared library typically has a private copy of the static data and shares the code.

The shared library design provides both static and dynamic benefits. The primary static benefit is reduced application file size by extracting common portions into shared libraries. The dynamic benefits include improved functional coherence by having all applications use one version of a library, the ability to upgrade common portions without modifying all applications, and improved RAM utilization by sharing code.

Fully generalized symbol resolution at runtime is not supported. This model is oriented more towards benefits for the end user than the programmer. Choices which reduce problems for the consumer are taken over generality for the programmer. In particular the approach to dynamic binding allows good runtime performance and fewer surprises by limiting the size of name spaces and the amount of relocation performed. The runtime actions are to bind existing symbolic linkages to addresses. The symbolic linkages are determined in a traditional manner at link time.

Having imports name both the library and symbol improves lookup performance and reduces possible confusion. The TOC model already condenses the references across execution units, reducing the amount of relocation to perform at load time. Further simplification is obtained by reducing the visible name space of libraries and resolving external references to specific libraries at link time. The name space is reduced when the shared library is linked. Exports must be explicitly selected.

When a fragment is linked some form of the shared library must be available and given to the linker just like a traditional linker library. Language level external references to routines and data are created as always. If the linker resolves a reference to a name exported from a shared library it creates information for the loader noting the use of that name in that library and relocation information for the reference. The runtime loader will lookup that name in that library and fill in the appropriate address. This adds security since the application will not become erroneously connected to some totally unrelated library on the user's machine that happened to export the same name.

An API for the Code Fragment Manager will be openly available. This allows programmers to utilize PowerPC code fragments from files or resources in any manner they choose. A single fragment may be loaded multiple times, each creating a different instance of the fragment with a new static data area. Addresses for individual routines and data may then be looked up and accessed through normal routine or data pointers. This lookup is within the context of a specific instance of a fragment, necessary to properly find exported data items and to ensure that calls (through descriptors) use the proper static data instance. This greatly expands the power of software components over the single–entry–point–no–static–data model in use for the 680x0.

Version Checking

Bidirectional version checking is supported between a shared library and its clients. This gives robust, automatic, overall compatibility checking at low cost. Other schemes such as Gestalt or a version number for each export either require manual intervention and/or much higher cost. Of course a Gestalt-like scheme could be used in addition to the automatic checking. Like any version number scheme, the success of it depends heavily on the proper use of version numbers by library developers.

For the sake of discussion we'll focus on an application using one shared library. The checking is actually done between any PowerPC fragment and each of the shared libraries that it imports from. The checking is entirely automatic, being performed as part of the runtime loading process.

When the application is created it is linked with a version of the shared library. Symbols in the shared library that are visible to the outside world are called exports. Unresolved external symbols in the application that are resolved by the linker to exports from the shared library are called imports. The version of the shared library used at link time is called the definition version. It supplies the definitions of symbols (the API), not the actual implementation of routines and variables (the code).

When the application is executed it must be connected to a version of the shared library. The imports in the application are connected to the associated exports in the shared library. This version of the shared library is called the implementation version. It supplies the actual implementation of routines and variables (the code).

There is a contract between the application and the shared library that the version of the shared library used at runtime must be compatible with the version of the shared library used at link time. The code must satisfy the API.

When the application is linked version information is copied from the shared library to the application. When the application is executed version information from the application and shared library are compared. This comparison is bidirectional, looking at the expectations of both the application and shared library.

The bidirectional nature of the checking is necessary because the definition version of the shared library might be newer or older than the implementation version. The basic approach is to let whoever is newer decide if they are compatible, on the premise that understanding the past is easier than predicting the future.

Let's suppose the application is linked with version 3 of the shared library and imports routine foo. If the application is run on a machine with version 1 of the library, foo might not exist at all or might have an incompatible implementation. (Yes, we would detect the missing foo anyway, the version check does it much more quickly if many exports are involved.) When version 3 of the library was created the library developer should know what older implementations (code) could satisfy the current definition (API).

——— Anticipatory Note ———

Actually imports are allowed to go unmatched at load time in very specific circumstances. These are described later.

If the application is run on a machine with version 5 of the library, an analogous situation holds. Foo might have been removed or have changed in an incompatible manner. When version 5 of the shared library was created the library developer should know what older definitions (API) could be satisfied by the current implementation (code).

Every shared library has three version numbers, the current version (Current), the old definition version (Old_API), and the old implementation version (Old_Code). These numbers must be set by the shared library developer when creating the shared library. Old_API and Old_Code must always be less than or equal to Current.

The value for Current ought to be obvious. Of course if the application is linked and executed with the same version of the shared library everything is assumed to be OK.

The value for Old_API is the oldest version for which the current implementation (code) is still compatible with the old definition (API). It says whose old expectations are still being satisfied. An old application linked with a definition of the shared library whose version is greater than or equal to

Old_API (and implicitly less than Current) is assumed to be compatible and may execute with the current (newer) implementation of the shared library.

The value for Old_Code is the oldest version for which the current definition (API) is still compatible with the old implementation (code). It says who can still satisfy today's expectations. An old implementation of the shared library with a version number greater than or equal to Old_Code (and implicitly less than Current) is assumed to be compatible and may execute with applications linked with the current definition of the shared library.

The linker copies the Current and Old_Code values from the definition library into the application. At runtime these values from the application are used along with the Current and Old_API values from the implementation library. Again, these values are actually copied and tested separately for each shared library used. The following C pseudo-code shows the tests are made at runtime:

```
if Definition.Current == Implementation.Current
   all OK
else if Definition.Current > Implementation.Current
   // Have new definition (API) with old implementation (code)
   if Definition.Old_Code <= Implementation.Current
      all OK
   else
      the library is too old for the application
else // Implementation.Current > Definition.Current
   // Have new implementation (code) with old definition (API)
   if Implementation.Old_API <= Definition.Current
      all OK
   else
      the application is too old for the library
```

Open Issue

The exact format of version numbers has not been settled.

Creating and Using Shared Libraries

Supporting this model means that the shared library developer must do some up front work for the application developer. The library developer must provide some form of header files for the exported names and at least a definition library exporting those names. The definition library need not be a fully operational version. The application developer needs only a library exporting the correct names to link the application. Working versions of the shared library are only needed for executing the application.

Explanatory Note

This section generally mentions applications and shared libraries. This is done only to enforce a clear mental model of the client of a service (the application) and the provider of the service (the shared library). As discussed earlier, all fragments are first class citizens and may be clients of shared libraries. Shared libraries are by definition those things used at link time and automatically found at runtime. Not all fragments are shared libraries.

The shared library is created in a normal link operation, specifying that the output is a shared library instead of an application and what the exported names are. Note that a shared library containing only data is perfectly acceptable. A shared library containing only code can be built and managed by CFM, but it is a morally suspect creature. If it only has code, it does not have the standard routine descriptors since they require relocation and hence go in a data section.

The application developer first obtains the headers and definition library. The headers are used in compiling the application in the time honored fashion. The definition library is used when linking the application, exactly as one uses "traditional" linker libraries. The definition library may be located anywhere in the file system, it does not have to be at the same location as the implementation library used at runtime. The linker records in the application a simple name for the shared library, the version numbers, and imported symbol names.

When ready to test, the application developer must obtain a working implementation of the shared library. This can be placed in the same folder as the application, or within the system folder much like extensions. When launching an application the necessary shared libraries are looked for first in the application folder then within the system folder. The search looks for the libraries by their simple name and performs the version checks. If an incompatible version is found the search continues, allowing multiple versions to coexist.

——— Open Issue ———

There are some open issues regarding the naming and lookup of shared libraries. We would like to satisfy the following goals:

- Shared library names are not bound to the file names
- Shared libraries can be found quickly at runtime
- Shared libraries can be found anywhere on the network
- Multiple versions can coexist and the "best" one found
- Shared libraries can receive "partial updates"

Partial updates allow a large shared library to be updated without complete replacement. Exports from the update library shadow corresponding exports from the original library.

The classical way to handle the first goal is to put the shared library name in a resource. This presents no problem for the linker. It could make lookup slow however, opening the resource forks for numerous files is a considerable burden.

The whole set of goals could be met by a robust, network aware shared library registry. However creating such a registry is beyond the scope of this one use. Ideally Apple should have a general registry for many things such as shared libraries, QuickTime components, system extensions like printer drivers, etc.

An intermediate, pragmatic approach could be based on the following:

- Shared libraries will have type "shlb" and a resource giving their name.
- The name will be captured from the resource by the linker.
- At boot, the CFM will scan specific folders to "register" shared libraries.
- The Finder may inform the CFM of other libraries added later.
- The CFM uses the following search rules:
    1. In the application folder by file name
    2. In the application folder by file type and resource
    3. In the "registry"
- Search all locations for an exact match first
- If no exact match is found search all locations for a compatible match
- If no match yet perform a wider search by file type and resource
- Finally abort the load.

The compatibility search could give preference to newer versions of the shared library, on the presumption that newer is better, less buggy, etc.

The specific folders scanned at boot are undefined. System software and human interface experts should determine whether shared libraries go in the current Extensions folder, a new Shared Libraries folder, or elsewhere. Subfolders should also be searched, allowing major developers to collect their libraries together.

Unmatched Imports at Runtime

All external symbols must be resolved at link time or the link will fail. Normally all imports must be matched at runtime or the load will fail. There are times, common in Macintosh programming, where it is necessary to be compatible with multiple versions of external software that have incompatible APIs. Explicitly coded checks are used to determine on–the–fly which API to use. The significant evolution of Macintosh system software has caused a lot of this. The TrapAvailable and Gestalt services were created to support the necessary checks.

As alluded to in an earlier note, the application developer may specify that certain imported symbols should be allowed to go unmatched at runtime. These imports are "tagged" at build time. Normally all imports must be matched to exports for the loading process to succeed. However if an unmatched import has been tagged the loading will succeed. The application is presumed to be making explicit checks before using such an imported routine or variable. A CFM service is available to check the status of any import by name. The standard Gestalt service will probably also be used in specific cases.

This feature does not circumvent the version checks. The versions must be compatible before an attempt is even made to match the imports.

——— Rationale Note ———

Forcing the application developer to specifically identify these special imports at build time improves the likelihood that the explicit availability checks are really made. This reduces the chance of obscure crashes on your mother's laptop while she is writing your evil twin out of her will.

——— Implementation Note ———

Using unresolved imports probably also means manually "tweaking" the Old_Code version number captured at link time. The tools to do this are defined elsewhere.

Fragment Instances in Memory

Earlier diagrams and discussion introduced the notions that code is always shared and that each shared library used by an application has a separate copy of its static data for that application. The shared library code is totally ignorant of this, as seen in the discussion of global addressing. Each copy of a fragment section in memory is an instance of that section.

Fragments are loaded into memory in specific contexts. A context defines the scope of a collection of interconnected fragments. Its primary effect is on the creation section instances. A separate context typically exists for each application (or Process Manager process, or whatever). The definition of a context is not hardwired into the CFM. The caller of the CFM, such as the Process Manager, defines a new context as necessary through a CFM service.

——— Explanatory Note ———

Flogging that horse: Every fragment's sections have instances and live in contexts, not just applications and shared libraries. Ya–dee–ya–dee–ya ...

The CFL API allows each memory section to have a separate sharing attribute. The most common form of sharing is context sharing. Each context will have at most one copy of a context shared section. Two extreme forms of sharing are global and never. At most one copy of a globally shared section exists on a machine. Pure, read–only sections like code should be globally shared. Sections that are never shared have a separate instance for each load.

——— Implementation Note ———

Additional levels of sharing are defined for microkernel supported tasks and teams.

——— Explanatory Note ———

A common example of context sharing occurs with standard language libraries like libc. If the application uses libc there will be a copy of libc's static data in the application's context. If the application then explicitly loads a fragment from a resource that also uses libc, the new fragment will be connected to the existing instance of libc.

Globally shared data sections should be used with great care. In particular shared libraries that have code sections should not have their static data set as globally shared. Since there would only be one instance of the shared library's TOC, it would only see one instance of its imports. This means that the full transitive closure of referenced libraries would be globally shared by implication. Globally shared data is best implemented as a data–only library.

——— Implementation Note ———

This provides a convenient future path for cleaning up low memory globals and their switching by the Process Manager. Truly global items would go in a data–only globally–shared library. Switched items would go in a context–shared library.

Never shared data is appropriate for fragments that are intended to be used multiple times in a context, with a fresh copy of static data each time. Examples include filters and protocol handlers that are written to have static variables maintaining information about a single data stream or connection.

Fragment Initialization and Termination

Any fragment can have an initialization and/or termination routine, not just shared libraries. But they are especially valuable for shared libraries and this discussion had to go somewhere.

As mentioned earlier, initialization, main, and termination routines may be optionally specified when building a fragment. The initialization and termination routines have comparable meaning for any fragment. The use of the main routine depends on the nature of the fragment.

The initialization routine is intended to be used for any necessary non-static initialization. One common use of this is to create self-initializing fragments. Instead of requiring all clients to invoke the InitMondoLib routine, the library developer can simply make InitMondoLib be the initialization routine. Another common use is for initializations required by programming languages. The best known example are C++ static objects, which require invocation of the appropriate constructors. Other languages allow static variables to be initialized with arbitrary expressions that are evaluated at runtime.

The termination routine is an obvious counterpart, allowing a fragment to clean up after itself. Typical uses include a place to call C++ destructors, to release system resources, to record statistics, etc.

——— Open Issue ———

There are coordination issues to be resolved in order to support a mixture of both language required initialization and hand–written initialization. Required initialization from multiple languages must also be supported. Language implementors must be open and document their required initialization and termination protocols, including external dependencies. Ideally this should be a pair of routine calls. This will allow developers to use those routines directly, or to wrap them in their own initialization and termination routines.

I.e. no mysterious, undocumented C++ munchers and _mains!

The initialization routine is called whenever a fragment is loaded, that is whenever an instance of it is created. If multiple instances of a fragment are created by direct calls to the CFM, the initialization routine will be called each time. The simple rule is that each time a new instance of a data section is created the initialization routine will be called.

If the loading of a fragment causes the loading of shared libraries that it needs then those shared libraries will of course be initialized too. If the fragment is connected to existing instances of the shared libraries then the libraries are not reinitialized.

When the loading of a fragment brings in new shared libraries, the complete set of libraries needed is first determined, then initialization ordering for the fragment and new libraries is determined. If fragment A depends on fragment B then an attempt is made to have B initialized before A. This cannot be done in the case of mutual dependencies. Longer circularities must also be detected, for example A requires B, B requires C, and C requires A. The default resolution of circularities is to arbitrarily pick which one is initialized first.

In many cases of mutual or circular dependencies the initialization ordering will not in fact matter. The initialization order only matters when the <u>initialization</u> of A depends on the initialization of B. In other words if the initialization code for A is using B in a manner that requires B to have been initialized.

To handle these cases, a fragment may specify which of the libraries that it depends on must be initialized first. This is done at build time by simply naming those special libraries. Circularities in required initializations will cause the load to fail.

——— Explanatory Note ———

Again, the initialization only includes the newly loaded libraries. If A depends on B and B is already loaded then B is assumed to be initialized. When a library is loaded it is connected to a specific instance of the libraries that it needs. If A and B are mutually dependent where B <u>must</u> be initialized first, and brought in together, then they will be connected together and initialized properly. If a new instance of B is brought in to the same context it will be connected to the old instance of A. The old instance of A will still be connected to the old instance of B. Only the new B will be initialized, after the A to which it is connected.

——— Rationale Note ———

The approach in the above note is reasonable when you consider that it is A who specifies that B must be initialized first, because A uses B in its own initialization. B says nothing. A is in fact connected to a copy of B that was initialized before it. The design is robust, and implementable. The example above is pathological and used only to illustrate the point of which initializations are called when.

XI. PowerPC Primitive Types

Integer Types

The PowerPC provides only unsigned (zero-fill) loads for byte (8-bit) integers. Both signed and unsigned loads are provided for halfword (16-bit) integers. A 64-bit CPU supports both signed and unsigned loads for word (32-bit) integers along with loads for doubleword (64-bit) integers. By signed and unsigned is meant only what happens to the high order bits of the register on a load. They are never left alone. Loads never set condition codes, so that aspect of sign is not an issue.

—— **Implementation Note** ——

The word load instruction of 32-bit CPUs does zero extension on a 64-bit CPU, not sign extension. This was chosen to preserve address semantics.

There are also instructions to sign extend bytes and halfwords already in a register, and words too on 64-bit CPUs. Zero extension is of done via the rotate-and-mask instructions.

All arithmetic and logical operations are done on values in registers. Both signed and unsigned comparisons are available.

Floating Point Types

The PowerPC supports single precision (32-bit) and double precision (64-bit) IEEE floating point formats in hardware. A 128-bit format will be supported in software, composed of two double precision values. The 128-bit format is anticipated to have "reasonable" performance. For backward compatibility 80-bit and 96-bit extended formats from the 680x0 world will be supported at some level by system software. These two formats are anticipated to have performance considerably worse than the 128-bit format. Details on all three software formats are found elsewhere.

XII. PowerPC Coding Conventions

This section does not cover high level language source conventions but code generation conventions that compilers and assembly language programmers should follow.

Data Alignment

The PowerPC hardware architecture supports unaligned accesses, but allows them to be slow.

The default type alignment rule is that scalar types are aligned to their size. That is words go on 32 bit boundaries, long floats go on 64 bit boundaries, etc. Composite types are aligned according to their most restrictive component. An array is aligned according to the element type and a structure is aligned according to the largest scalar it contains directly or in a substructure. This alignment is applied by compilers when laying out composite types and data areas such as stack frames. They are applied by the linker (under the compiler's direction) when laying out static data areas.

Compiler pragmas should be provided to pack structure fields on byte and halfword boundaries. The halfword packing is necessary to build 680x0 compatible structures without introducing visible pad fields. Compilers are free to rearrange unpacked structures to minimize the holes, they must not rearrange packed structures.

Global Item Naming Rules

Some of the rules for the naming of routines (code and descriptors), static data, and TOC entries have already been given in examples. These rules are taken from AIX on the RS/6000 for compatibility.

XCOFF has a notion of CSECT, much like the module in the MPW object module format. It is the unit in which clusters of code or data are included in a link. CSECTs have a class which reflects their use; common classes are PR (code), DS (routine descriptor), RW (static variable), and TC (TOC entry). CSECTs are denoted by their name and class as Name[Class] or Name{Class}. In effect the class is part of the name and is used in reference matching by the linker.

Routines conventionally have a code label which is the routine name prefixed with a period. They have a descriptor in a DS CSECT whose name is the unadorned routine name. Calls to external routines are compiled as calls to the external symbol ".name[PR]". If the linker cannot resolve the symbol it will look for a shared library export of the form "name[DS]".

Global variables (C "extern" variables) are each put in a separate RW CSECT with the name of the variable.

TOC pointers are each in a separate TC CSECT with the name of the item they are pointing to. This is important only for the folding of duplicate TOC pointers, not for their proper resolution.

——— Picture Place ———

Put in some assembly language examples. Explain the .tc macro.

Using 64–bit CPUs

The PowerPC hardware architecture is defined in terms of a 64–bit model and 32–bit and 64–bit CPU implementations. There are actually three classes of execution:

- Execution on a 32–bit CPU. In this case 64–bit instructions are not available.
- Execution in 32–bit mode on a 64–bit CPU. Addresses are still 32–bits, but 64–bit instructions are available for data movement. Most arithmetic works properly, except for the use of the Rc bit.
- Execution in 64–bit mode on a 64–bit CPU.

This software architecture does not define a full 64–bit model. Doing this properly requires system software support for addressing. Allowing arbitrarily intermingled 32–bit and 64–bit routines is difficult because of the split use of caller's and callee's stack frames. Rules are given to allow software to take limited advantage of 64–bit CPUs:

- There will be no support for 64–bit addressing in the defined future. Software running in 64–bit mode must ensure that all addresses stay in the range of 0 to 2**32. (Be sure to use full 64–bit values for negative offsets!)

- Operation in 64–bit mode will be interrupt and task switch safe.

- Routine calls must all be made in 32–bit mode. The O/S will ensure this for callbacks. Callers using the high order part of a non-volatile register must save it themselves.

- The Apple C and C++ compilers may provide options to generate 64–bit data movement instructions and/or 64–bit arithmetic. In the latter case a "long long" integer type will be supported. The existing integer types will keep their present sizes.

Handling Optimization

Compilers must recognize that 680x0 style heap compaction may occur and avoid generating code that would fail in this case. For example the left hand side of assignments should be evaluated after the right hand side. Aliasing must be carefully (properly) implemented to avoid erroneously keeping dereferenced handles in registers. This will help avoid the problems illustrated by:

```
**handle.field = function_causing_compaction;

proc (func1_causing_compaction(**handle.field1),
      func2_causing_compaction(**handle.field2));
```

Open work:

Add a glossary

Mention special restrictions on native resources, namely expansion of BSS.

Add discussion of traceback tables

Document the PowerPC patching model here

Discuss extra possible sections in containers for language exception handling or debugging

Document "update" libraries here (at least a little)