EARLY MACINTOSH TECHNICAL INFORMATION

INSIDE MACINTOSH
PUTTING TOGETHER
A MACINTOSH APPLICATION

COMMENT

MACINTOSH PROGRAM DEVELOPMENT INSTRUCTIONS FOR MACINTOSH PROGRAMMERS USING THE APPLE LISA WORKSHOP DEVELOPMENT SYSTEM

AUTHOR

APPLE COMPUTER

DATE

05 MAY 1985

SOURCE

DAVID T CRAIG • JANUARY 2004

MACINTOSH USER EDUCATION

Putting Together a Macintosh Application /PUTTING/TOGETHER

Modification History:	First Draft (ROM 2.45)	Caroline Rose	6/9/83
	Second Draft (ROM 4.4)	Caroline Rose	7/14/83
	Third Draft (ROM 7)	Caroline Rose	1/13/84
	Fourth Draft	Caroline Rose	4/9/84
	Fifth Draft	Caroline Rose	7/10/84
	Sixth Draft	Caroline Rose	5/5/85

ABSTRACT

This manual discusses the fundamentals of preparing, compiling or assembling, and linking a Macintosh application program on the Lisa Workshop development system.

Summary of significant changes and additions since last draft:

- This manual now documents Lisa Workshop version 3.0 and the May 1985 Macintosh Software Supplement. Some of the information may not apply to Workshop version 2.0.

- Changes have been made to the interface files and the files you link with or include in your assembly-language source.

- The sections describing the Macintosh utility programs RMover and Set File have been removed. These programs have been superseded by other tools in the Macintosh Software Supplement.

2 Putting Together a Macintosh Application

TABLE OF CONTENTS

ABOUT THIS MANUAL

This manual discusses the fundamentals of preparing, compiling or assembling, and linking a Macintosh application program on the Lisa Workshop development system. It assumes the following:

- You know how to write a Macintosh application in Pascal or assembly language. Details on this may be found in Inside Macintosh.

- You're familiar with the Macintosh Finder, which is described in Macintosh, the owner's guide.

You need to have a Lisa 2/5 or 2/10 with at least 1 megabyte of memory, a Workshop development system (version 2.0 or greater), and the Macintosh Software Supplement.

(note)
This manual applies to version 3.0 of the Workshop and the May 1985 Software Supplement.

After explaining some conventions it uses, the manual begins by presenting the first steps you should take once your Lisa has been set up for Macintosh application development under the Workshop. It then discusses each of the three files you'll create to develop your application: the source file, the Resource Compiler input file, and an exec file.

The next section discusses how to divide an application into segments. This is followed by important information for programmers who want to write all or part of an application in assembly language.

Finally, there's a summary of the steps to take to put together a Macintosh application.

(note)
This manual presents a recommended scenario, not by any means the only possible one. Details, such as what you name your files, may vary.

Conventions

Sometimes this manual shows you what to do in a two-column table, the first one labeled "Prompt" and the second "Response". The first column shows what appears on the Lisa to "prompt" you; it might be a request for a file name, or just the Workshop command line. This column will not show all the output you'll get from a program, only the line that prompts you. (There may have been a lot of output before that line.) The second column shows what you type as a response. The following notation is used:

Notation	Meaning
<ret>	Press the RETURN key.
[ ]	Explanatory comments are enclosed in [ ]; you don't type them.

A space preceding <ret> is not to be typed. It's there only for readability.

[ ] in the "Prompt" column actually appear in the prompt; they enclose defaults.

Except where indicated otherwise, you may type letters in any combination of uppercase and lowercase, regardless of how they're shown in this manual.

GETTING STARTED

Once your Lisa has been set up for Macintosh application development, it's a good idea to orient yourself to the files installed on it. You can use the List command in the File Manager to list all the file names. Certain subsets of related files begin with the same few letters followed by a slash; some typical naming conventions are as follows:

Beginning of file name	Description
Intrfc/	Text files containing the Pascal interfaces
TlAsm/	Text files to include when using assembly language
Obj/	Object files
Work/	Your current working files
Back/	Backup copies of your working files
Example/	Examples provided by Macintosh Technical Support

(note)
This manual assumes that your files observe the above naming conventions.

You'll write your application to a Macintosh system disk, which means a Macintosh disk that contains the system files needed for running an application. The necessary system files are on the Mac Build disk that you received as part of the Macintosh Software Supplement. Use that disk only to create other system disks. Here's how:

1. Insert the Mac Build disk into the Macintosh and open it.

2. Copy the System Folder to a new Macintosh disk; the exact method you use depends on whether you have an external drive. See the Macintosh owner's guide for more information.

(note)
One of the files in the System Folder, Imagewriter, is needed only if you're going to print to an Imagewriter

printer; to save space, you might not want to copy it if you don't need it.

If you also need or want any of the files on the MacStuff disks included in the Macintosh Software Supplement, copy them as well.

As described in detail in the following sections, you'll create a source file, Resource Compiler input file, and exec file for your application, insert your Macintosh system disk into the Lisa, and run the exec file. The exec file will compile the source file, link the resulting object file with other required object files, run the Resource Compiler to create the application's resource file, and run a program called MacCom to write the application to the Macintosh disk. When MacCom is done, it will eject the disk; to try out your application, you'll insert the ejected disk into the Macintosh and just open the application's icon.

THE SOURCE FILE

Your working files will of course include the source file for your application. Suppose, for example, that you have an application named Samp. The source file would be Work/Samp.Text and would have the structure shown below.

(note)
"Samp" is used as the application name in all examples in this manual. You don't have to use the exact name of your application; any abbreviation will do.

```
PROGRAM Samp;

{ Samp -- A sample application written in Pascal }
{         by Macintosh User Education  5/1/85    }

[ List the following in the order shown. ]

USES {$U Obj/MemTypes    } MemTypes,
     {$U Obj/QuickDraw   } QuickDraw,
     {$U Obj/OSIntf      } OSIntf,
     {$U Obj/ToolIntf    } ToolIntf,
     {$U Obj/MacPrint    } MacPrint,    [ OPTIONAL ]
     {$U Obj/SANELib     } SANELib,     [ OPTIONAL ]
     {$U Obj/PackIntf    } PackIntf;    [ OPTIONAL ]

[ Your LABEL, CONST, TYPE, and VAR declarations will be here. ]

[ Your application's procedures and functions will be here. ]

BEGIN

  [ The main program will be here. ]

END.
```

Each line in the USES clause specifies first a file name and then a unit name (which happen to be the same in all cases here). The file contains the compiled Pascal interface for that unit; the corresponding text file name begins with "Intrfc/" rather than "Obj/". The Pascal interface includes the declarations of all the routines in the unit. It also contains any data types, predefined constants, and, in the case of QuickDraw, Pascal global variables.

File name	Interface it contains
Intrfc/MemTypes.Text	Basic Memory Manager data types
Intrfc/QuickDraw.Text	QuickDraw
Intrfc/OSIntf.Text	Operating System
Intrfc/ToolIntf.Text	Toolbox, except QuickDraw
Intrfc/MacPrint.Text	Printing Manager
Intrfc/SANELib.Text	Floating-Point Arithmetic and Transcendental Functions Packages
Intrfc/PackIntf.Text	Other packages

You only have to include the files for the units your application uses. It doesn't do any harm to include them all, but it will take somewhat longer for your program to compile. If you're using any units of your own, just add their Pascal interface files at the end of the USES clause.

You can divide the code of an application into several segments and have only some of them in memory at a time. The section "Dividing Your Application Into Segments" tells how to specify segments in your source file. If you don't specify any, your program will consist of a single, blank-named segment.

THE RESOURCE COMPILER INPUT FILE

You'll need to create a resource file for your application. This is done with the Resource Compiler, and you'll have among your working files an input file to the Resource Compiler. One convention for naming this input file is to give it the name of your source file followed by "R" (such as Work/SampR.Text).

The first entry in the input file specifies the name to be given to the output file from the Resource Compiler, the resource file itself; you'll enter "Work/" followed by the application name and ".Rsrc". Another entry tells which file the application code segments are to be read from. (The code segments are actually resources of the application.) You'll enter the name of the Linker output file specified in the exec file for building your application, as described in the next section.

8 Putting Together a Macintosh Application

If you don't want to include any resources other than the code segments, you can have a simple input file like this:

```
*  SampR -- Resource input for sample application
*           Written by Macintosh User Education  5/1/85

Work/Samp.Rsrc

Type SAMP = STR
  ,Ø
Samp Version 1.1 -- May 1, 1985

Type CODE
  Work/SampL,Ø
```

This tells the Resource Compiler to write the resulting resource file to Work/Samp.Rsrc and to read the application code segments from Work/SampL.Obj. It also specifies the file's signature and version data, which the Finder needs.

It's a good idea to begin the input file with a comment that describes its contents and shows its author, creation date, and other such information. Any line beginning with an asterisk (*) is treated as a comment and ignored. (You cannot have comments embedded within lines.) The Resource Compiler also ignores the following:

- leading spaces (except before the text of a string resource)
- embedded spaces (except in file names, titles, or other text strings)
- blank lines (except for those indicated as required)

The first line that isn't ignored specifies the name to be given to the resulting resource file. Then, for each type of resource to be defined, there are one or more "Type statements". A Type statement consists of the word "Type" followed by the resource type (without quotes) and, below that, an entry of following format for each resource:

```
file name!resource name,resource ID (resource attributes)
type-specific data
```

The punctuation shown here in the first line is typed as part of the format. Don't enter spaces where none are shown, such as after the comma. You must always provide a resource ID. Specifications other than the resource ID may or may not be required, depending on the resource type:

- Either there will be some type-specific data defining the resource or you'll give a file name indicating where the resource will be read from. Even in the absence of a file name, you must include the comma before the resource ID.

- You specify a resource name along with the file name for fonts and drivers. The Menu Manager procedures AddResMenu and InsertResMenu will put these resource names in menus. Enter the names in the combination of uppercase and lowercase that you want to appear in the menus.

- Resource attributes in parentheses are optional for all types. They're given as a number equal to the value of the resource attributes byte, and Ø is assumed if none is specified. For example, for a resource that's purgeable but has no other attributes set, the input will be "(32)".

If you want to enter a nonprinting or other unusual character in your input file, either by itself or embedded within text, just type a back slash (\) followed by the ASCII code of the character in hexadecimal. For example, the Resource Compiler interprets \ØD as a Return character and \14 as the apple symbol.

The formats for the different types of resources are best explained by example. Some examples are given below along with remarks that provide further explanation. Here are some points to remember:

- Most examples list only one resource per Type statement, but you can include as many resources as you like in a single statement.

- In every case, resource attributes in parentheses may be specified after the resource ID.

- All numbers are base 1Ø except where hexadecimal is indicated.

- The Type statements may appear in any order in the input file.

```
Type WIND                       Window template
  ,128 (32)                     Resource ID
  Status Report                 Window title
  4Ø 8Ø 12Ø 3ØØ                 BoundsRect (top left bottom right)
  Visible GoAway                For FALSE, use Invisible or NoGoAway
  Ø                             ProcID (window definition ID)
  Ø                             RefCon (reference value)

Type MENU                       Menu, standard type
  ,128 (1)                      Resource ID (becomes the menu ID)
* menu for desk accessories
  \14                           Menu title (apple symbol)
    About Samp...               Menu item
                                Blank line required at end of menu
  ,129                          Resource ID
  Edit                          Menu title
    Cut/X                       Menu items, one per line, with meta-
    Paste/Z                      characters, ! alone for check mark
    (-                          You cannot specify a blank item; use (-
    Word Wrap!                   for a disabled continuous line.
                                Blank line required at end of menu
```

```
Type MENU                       Menu, nonstandard type
  ,2ØØ                          Resource ID   [ SEE NOTE 1 BELOW ]
  2Ø1                           Resource ID of menu definition procedure
  Patterns                      Menu title (may be followed by items)
                                Blank line required at end of menu
Type CNTL                       Control template
  ,128                          Resource ID
  Help                          Control title
  55 2Ø 75 9Ø                   BoundsRect
  Visible                       For FALSE, use Invisible
  Ø                             ProcID (control definition ID)
  1                             RefCon (reference value)
  Ø Ø Ø                         Value minimum maximum

Type ALRT                       Alert template
  ,128                          Resource ID
  12Ø 1ØØ 19Ø 25Ø               BoundsRect
  3ØØ                           Resource ID of item list
  F721                          Stages word in hexadecimal

Type DLOG                       Dialog template
  ,128                          Resource ID
* modal dialog
  1ØØ 1ØØ 19Ø 25Ø               BoundsRect
  Visible 1 NoGoAway Ø          1 is procID, Ø is refCon
  2ØØ                           Resource ID of item list
                                Title (none in this case)
  ,129
* modeless dialog
  1ØØ 1ØØ 19Ø 25Ø               BoundsRect
  Visible Ø GoAway Ø            Ø procID, Ø refCon
  3ØØ                           Resource ID of item list
  Find and Replace              Title

Type DITL                       Item list in dialog or alert
  ,2ØØ                          Resource ID
  5                             Number of items
  BtnItem Enabled               Also:  ChkItem, RadioItem
  6Ø 1Ø 8Ø 7Ø                   Display rectangle
  Start                         Title
                                Blank line required between items
  ResCItem Enabled              Control defined in control template
  6Ø 3Ø 8Ø 1ØØ                  Display rectangle
  128                           Resource ID of control template

  StatText Disabled             Also:  EditText
  1Ø 93 26 13Ø                  Display rectangle
  Seed                          The text (may be blank if EditText)

  IconItem Disabled             Also:  PicItem
  1Ø 24 42 56                   Display rectangle
  128                           Resource ID of icon
```

```
  UserItem Disabled          Application-defined item
  2Ø 5Ø 6Ø 85                Display rectangle

Type ICON                    Icon
  ,128                       Resource ID
  Ø38Ø ØØØØ                  The icon in hexadecimal (32 such lines
    . . .                     altogether)
  1ECØ 318Ø

Type ICN#                    Icon list
  ,128                       Resource ID
  2                          Number of icons
  ØØØ1 ØØØØ                  The icons in hexadecimal (32 such lines
    . . .                     altogether for each icon)
  ØØØ2 8ØØØ

Type CURS                    Cursor
  ,3ØØ                       Resource ID
  7FFC . . . 287F            The data:  64 hex digits on one line
  ØFCØ . . . 1FF8            The mask:  64 hex digits on one line
  ØØØ8 ØØØ8                  The hotSpot in hexadecimal (v h)

Type PAT                     Pattern
  ,2ØØ                       Resource ID
  AADDAA66AADDAA66           The pattern in hexadecimal

Type PAT#                    Pattern list
  ,136                       Resource ID
  2                          Number of patterns
  5522552255225522           The patterns in hexadecimal, one per
  FFEEDDCCFFEEDDCC            line

Type STR                     String
  ,128                       Resource ID
This is your string          The string on one line (leading spaces
                              not ignored)
Type STR#                    String list
  ,129                       Resource ID
First string                 The strings
Second string
* note Return in next string
Third string\ØDcontinued

                             Blank line required after last string
Type DRVR                    Desk accessory or other device driver
  Obj/Monkey!Monkey,17 (32)  File name!resource name,resource ID
                              [ SEE NOTE 2 BELOW ]
Type FREF                    File reference
  ,128                       Resource ID
  APPL  Ø  TgFil             File type  local ID of icon  file name
                              (omit file name if none)
```

12 Putting Together a Macintosh Application

```
Type BNDL                       Bundle
  ,128                          Resource ID
  SAMP  Ø                       Bundle owner
  2                             Number of types in bundle
  ICN#  1                       Type and number of resources
  Ø  128                        Local ID Ø maps to resource ID 128
  FREF  1                       Type and number of resources
  Ø  128                        Local ID Ø maps to resource ID 128

Type FONT                       Font (or FWID for font widths)
  Obj/Griffin!Griffin,4ØØ@Ø     File name!resource name,resource ID
  Obj/Griffin1Ø,4ØØ@1Ø          File name,resource ID   [ SEE NOTE 3 ]
  Obj/Griffin12,4ØØ@12          File name,resource ID   [ BELOW       ]

Type CODE                       Application code segments
  Obj/SampL,Ø                   Linker output file name,resource ID
                                 [ SEE NOTE 4 BELOW ]
```

Notes:

1. Notice that the input for a nonstandard menu has one extra line in it: the resource ID of the menu definition procedure, just following the resource ID of the menu. If that line is omitted (that is, if the menu's resource ID is followed by a line containing text rather than a number), the resource ID of the standard menu definition procedure (Ø) is assumed.

2. The Resource Compiler adds a NUL character (ASCII code Ø) at the beginning of the name you specify for a 'DRVR' type of resource. This inclusion of a nonprinting character avoids conflict with file names that are the same as the names of desk accessories.

3. The resource ID for a font resource has a special format:

   font number @ size

   The actual resource ID that the Resource Compiler assigns to the font is

   (128 * font number) + size

   Three font resources are listed in the example above. Size Ø is used to provide only the name of the font (Griffin in this case); a file name must also be specified but is ignored. The two remaining font resources define the Griffin font in two sizes, 1Ø and 12.

4. For a 'CODE' type of resource, ".Obj" is appended to the given file name, and the resource ID you specify is ignored. The Resource Compiler always creates two resources of this type, with ID numbers Ø and 1, and will create additional ones numbered sequentially from 2 if your program is divided into segments.

The Type statement for a resource of type 'WDEF', 'MDEF', 'CDEF', 'FKEY', 'KEYC', 'PACK', or 'PICT' has the same format as for 'CODE': Only a file name and a resource ID are specified. For the 'PICT' type, the file contains the picture; for the other types, it contains the compiled code of the resource, and the Resource Compiler appends ".Obj" to the file name.

(note)
The 'MBAR' resource type is not recognized by the Resource Compiler.

If your application is going to write to the resulting resource file as well as read it, you should place the Type statement for the code segments at the end of the input file. In general, any resources that the application might change and write out to the resource file should be listed first in the input file, and any resources that won't be changed (like the code segments) should be listed last. The reason for this is that the Resource Compiler stores resources in the reverse of the order that they're listed, and it's more efficient for the Resource Manager to do file compaction if the changed resources are at the end of the resource file.

Defining Your Own Resource Types

You can use one of the three types GNRL, HEXA, and ANYB to define your own types of resources in the Resource Compiler input file. GNRL allows you to specify your resource data in the manner best suited to your particular data format; you specify the data as you want it to appear in the resource. A code (beginning with a period) tells the Resource Compiler how to interpret what you enter on the next line or lines (up to the next code or the end of the Type statement). The following illustrates all the codes:

```
Type GNRL                    General type
  ,128                       Resource ID
  .P                         Pascal strings (with length byte), one
A Pascal string               per line
Another Pascal string
  .S                         Strings without length byte, one per
A string                      line
  .I                         Integers (decimal), one per line
  Ø
  1
  .L                         Long integers (decimal), one per line
  5438
  .H                         Bytes in hexadecimal, any number
  526FEEC942E78EA4            total, any number per line
  ØF4C
  .B                         Bytes from a file
  MyData  36  256            File name  number of bytes  offset
                             Blank line required at end of statement
```

You can use an equal sign (=) along with the GNRL type to define a

resource of any desired format and with any four-character resource type; for example, to define a resource of type 'MINE' consisting of the integer 57 followed by the Pascal string 'Finance charges', you could enter this:

```
Type MINE = GNRL
  ,400
  .I
  57
  .P
Finance charges
```

The Resource Manager call GetResource('MINE',400) would return a handle to this resource.

The types HEXA and ANYB simply offer alternatives to the .H and .B options (respectively) of the GNRL type, as shown below.

```
Type HEXA                       Bytes in hexadecimal
  ,201                          Resource ID
  526FEEC942E78EA4              The bytes (any number total, any
  0F4C                           number per line)
                                Blank line required at end
Type ANYB                       Bytes from a file
  MyData,200                    File name,resource ID
  36  256                       Number of bytes  offset in file
```

You can also define a new resource type that inherits the properties of a standard type. For example,

```
Type XDEF = WDEF
```

defines the new type 'XDEF', which the Resource Compiler treats exactly like 'WDEF'. The next line would contain a file name and resource ID just as for a 'WDEF' resource.

THE EXEC FILE

It's useful for each application to have an exec file that does everything necessary to build the application, including compiling, linking, creating the resource file, and writing to a Macintosh disk. The name of the exec file might, for example, be the source file name followed by "X" (for "eXec"). Work/SampX.Text, the exec file for the Samp application, is shown below.

```
$EXEC
P{ascal}$M+
Work/Samp
{no list file}
{default output file}
L{ink}?
+X
{no more options}
Work/Samp
Obj/QuickDraw
Obj/OSTraps
Obj/ToolTraps
Obj/PrLink          [ OPTIONAL ]
Obj/SANELibAsm      [ OPTIONAL ]
Obj/PackTraps       [ OPTIONAL ]
Obj/PasInit
Obj/PasLib
Obj/PasLibAsm
Obj/RTLib
{end of input files}
{listing to console}
Work/SampL
R{un}RMaker
Work/SampR
R{un}MacCom
F{inder info}Y{es}L{isa->Mac}Work/Samp.Rsrc
Samp
APPL
SAMP
{no bundle bit}
E{ject}Q{uit}
$ENDEXEC
```

The file begins with $EXEC and ends with $ENDEXEC. Everything in between (except for comments in braces) is exactly what you would type on your Lisa if you were not using an exec file. To show what the various entries in this file accomplish, the table below indicates what each of them is a response to, and shows your response as it is in the exec file or as it would be if you were using the keyboard. The numbers on the left are given for reference in the explanation that follows the table.

	Prompt	Response
[1]	Workshop command line	P [for Pascal]
	Input file - [.TEXT]	Work/Samp <ret>
	List file - [.TEXT]	<ret> [for none]
	Output file - [Work/Samp][.OBJ]	<ret> [for Work/Samp.Obj]

```
[2]  Workshop command line                    L  [for Link]
     Input file [.OBJ] ?                      ? <ret>  [for options]
     Options ?                                +X <ret>
     Options ?                                <ret>  [no more options]
     Input file [.OBJ] ?                      Work/Samp <ret>
     Input file [.OBJ] ?                      Obj/QuickDraw <ret>
     Input file [.OBJ] ?                      Obj/OSTraps <ret>
           . . .                              [other input files]
     Input file [.OBJ] ?                      Obj/RTLib <ret>
     Input file [.OBJ] ?                      <ret>  [end of input files]
     Listing file [-CONSOLE] / [.TEXT]        <ret>  [for -CONSOLE]
     Output file ? [OBJ.]                     Work/SampL <ret>
[3]  Workshop command line                    R  [for Run]
     Run what program?                        RMaker <ret>
     Input file [sysResDef][.TEXT] -          Work/SampR <ret>
[4]  Workshop command line                    R  [for Run]
     Run what program?                        MacCom <ret>
     MacCom command line                      F  [for Finder info]
     Always prompt for the Finder info
      when writing to a Mac file?
      (Y or N) [No]                           Y  [for Yes]
     MacCom command line                      L  [for Lisa->Mac]
     Lisa files to write to Mac disk?         Work/Samp.Rsrc <ret>
     Copy to what Mac file?                   Samp <ret>
     Type? [????]                             APPL <ret>
     Creator? [????]                          SAMP <ret>
     Set the Bundle Bit? (Y or N) [No]        <ret>  [for No]
     MacCom command line                      E  [for Eject]
     MacCom command line                      Q  [for Quit]
```

Here's what you accomplish at each of the steps:

1. You compile the Pascal source code (Work/Samp.Text), resulting in an object file (Work/Samp.Obj).

2. You link the application's object file with other object files (resulting in the output file Work/SampL.Obj).

3. You run the Resource Compiler to create the application's resource file (Work/Samp.Rsrc, as specified in Work/SampR.Text, the input file to the Resource Compiler). Included in the resources are the application's code segments, which are read from the Linker output file.

4. You use the MacCom program to write the resource file to the Macintosh disk, giving the file the exact name you want your application to have. You set its file type to 'APPL' and its creator to the signature specified in the resource file. Since there's no bundle in Samp's resource file, you don't set the bundle bit. Finally, you ask MacCom to eject the disk.

The files linked with the application's object file in step 3 are described below. Most of them contain a trap interface, which is a set of small assembly-language routines that make it possible to call the

corresponding unit or units from Pascal. The files should be listed in the order shown. Specify the optional files only if your application uses the routines they apply to.

File name	Description
Obj/MemTypes.Obj	Basic Memory Manager data types
Obj/QuickDraw.Obj	Pascal interface to QuickDraw, needed so the Linker will know how many QuickDraw globals there are
Obj/OSTraps.Obj	Trap interface for the Operating System
Obj/ToolTraps.Obj	Trap interface for the Toolbox (except QuickDraw)
Obj/PrLink.Obj	The Printing Manager (except low-level)
Obj/PrScreen.Obj	The low-level Printing Manager routines; can be specified instead of PrLink
Obj/SANELibAsm.Obj	The Floating-Point Arithmetic and Transcendental Functions Packages
Obj/PackTraps.Obj	Trap interface for other packages
Obj/PasInit.Obj	Predefined Pascal routines, such as POINTER and ORD4
Obj/PasLib.Obj	
Obj/PasLibAsm.Obj	
Obj/RTLib.Obj	

Before running the Exec file, insert a Macintosh system disk into the Lisa. Run the exec file as follows:

Prompt	Response
Workshop command line	R [for Run]
Run what program?	<Work/SampX <ret>

When the disk is ejected, remove it and insert it into the Macintosh. To try out your application, just open its icon.

(warning)
If you don't set your application's file type and creator, either you won't be able to open its icon in the usual way, or a different application may start up when you do open it!

Notice that if you change the application's signature or the setting of its bundle bit, step 4 of the above exec file will have to be edited accordingly. Furthermore, if you create an icon for your application (or modify it), you'll have to delete the invisible Desktop file, otherwise the Finder won't know about the new icon. You can delete the Desktop file by using the Delete command in MacCom on the Lisa, just before copying the application to the disk with MacCom, or by holding down the Option and Command keys when you start up the system disk on the Macintosh.

(note)
Deleting the Desktop file can also affect the folder structure on the disk.

18 Putting Together a Macintosh Application

Before making major changes to your application, it's a good idea to back it up. You can use the Backup command in the File Manager to back up all files beginning with "Work/" to files beginning with "Back/" (Work/=,Back/=). Also, you might want to periodically back up your working files onto 3 1/2-inch disks.

There are several ways you could refine the exec file illustrated here; exactly what you do will depend on your particular situation. Some possibilities are listed below.

- You can set up the exec file to compile or link only if actually necessary. For more information, see your Workshop documentation or the sample general-purpose exec file (Example/Exec.Text) provided in the Macintosh Software Supplement.

- To save disk space, you can add commands to the exec file to make it delete the two intermediate files: the object file for the application and the Linker output file.

- If you want to keep the intermediate files around but are working on more than one application, you can save disk space by giving the intermediate files the same name for all applications (say, "Work/Temp").

- You can embed the exec file in your program's source file. To do this, you must use "(*" and "*)" around the exec part of the file and use the I invocation option. See your Workshop documentation for details.

DIVIDING YOUR APPLICATION INTO SEGMENTS

You can specify the beginning of a segment in your application's source file as follows:

```
{$S segname}
```

where segname is the segment name, a sequence of up to eight characters. Normally you should give the main segment a blank name. For example, you might structure your program as follows:

```
PROGRAM Samp;

[ The USES clause and your LABEL, CONST, and VAR declarations
  will be here. ]

{$S Seg1}

[ The procedures and functions in Seg1 will be here. ]

{$S Seg2}

[ The procedures and functions in Seg2 will be here. ]

{$S     }

BEGIN

  [ The main program will be here. ]

END.
```

You can specify the same segment name more than once; the routines will just be accumulated into that segment. To avoid problems when moving routines around in the source file, some programmers follow the practice of putting a segment name specification before every routine.

(warning)
Uppercase and lowercase letters are distinguished in segment names. For example, "Seg1" and "SEG1" are not equivalent names.

If you don't specify a segment name before the first routine in your file, the blank segment name will be assumed there.

NOTES FOR ASSEMBLY-LANGUAGE PROGRAMMERS

You can write all or part of your Macintosh application in assembly language. Suppose, for example, that you write most of it in Pascal but have some utility routines written in assembly language. Your working files will include a source file and object file for the assembly-language routines (say, Work/SampA.Text and Work/SampA.Obj). The source file will have the structure shown below.

```
; SampA -- Assembly-language routines for Samp
;          Written by Macintosh User Education  5/1/85

[ List the following in the order shown. ]

.INCLUDE TlAsm/SysEqu.Text
.INCLUDE TlAsm/SysTraps.Text
.INCLUDE TlAsm/SysErr.Text
.INCLUDE TlAsm/QuickEqu.Text
.INCLUDE TlAsm/QuickTraps.Text
.INCLUDE TlAsm/ToolTraps.Text
.INCLUDE TlAsm/ToolEqu.Text
.INCLUDE TlAsm/PrEqu.Text        [ OPTIONAL ]
.INCLUDE TlAsm/SANEMacs.Text     [ OPTIONAL ]
.INCLUDE TlAsm/PackMacs.Text     [ OPTIONAL ]
.INCLUDE TlAsm/FSEqu.Text        [ OPTIONAL ]

[ Here there will be a .PROC or .FUNC directive for each routine, ]
[ followed by the routine itself.  Two examples follow.  ]

; PROCEDURE MyRoutine (count: INTEGER);

.PROC MyRoutine

MyRoutine
      [ the code of MyRoutine ]

; FUNCTION MyOtherRoutine : LongInt;

.FUNC MyOtherRoutine

MyOtherRoutine
      [ the code of MyOtherRoutine ]

.END
```

(note)
The .PROC or .FUNC directive clears the symbol table, so symbols defined in one routine can't be referred to in another (without an explicit reference using .REF). If you want to share code between routines, you can instead have a single .PROC directive for SampA followed by a .DEF directive for each routine name.

Including unneeded files with .INCLUDE directives will do no harm except make your program take longer to assemble. The files marked as optional above are the least commonly needed; even some of the others may not be required. Here's what the files contain:

File name	Description
TlAsm/SysEqu.Text	System equates
TlAsm/SysTraps.Text	System traps
TlAsm/SysErr.Text	System error equates
TlAsm/QuickEqu.Text	QuickDraw equates
TlAsm/QuickTraps.Text	QuickDraw traps
TlAsm/ToolTraps.Text	Toolbox traps, except QuickDraw
TlAsm/ToolEqu.Text	Toolbox equates, except QuickDraw
TlAsm/PrEqu.Text	Equates for Printing Manager
TlAsm/SANEMacs.Text	Macros and equates for Floating-Point Arithmetic and Transcendental Functions Packages
TlAsm/PackMacs.Text	Macros and equates for other packages
TlAsm/FSEqu.Text	File system equates

If you've created any similar files for units of your own, just add .INCLUDE directives for them after the last .INCLUDE directive shown above.

To specify the beginning of a segment in assembly language, you can use the directive

```
.SEG 'segname'
```

where segname is the segment name, a sequence of up to eight characters.

For each assembly-language routine invoked from Pascal, the Pascal source file for your application will include an external declaration. For example:

```
PROCEDURE MyRoutine (count: INTEGER); EXTERNAL;
FUNCTION  MyOtherRoutine : LongInt; EXTERNAL;
```

If the routines form a unit that may be used by other applications, you should instead prepare a Pascal interface file for the unit and include it in the USES clause in the application's source file.

You'll assemble the Work/SampA.Text file as shown below.

Prompt	Response
Workshop command line	A [for Assemble]
Input file - [.TEXT]	Work/SampA <ret>
Listing file (<CR> for none) - [.TEXT]	<ret> [for none]
Output file - [Work/SampA] [.OBJ]	<ret> [for Work/SampA.Obj]

(note)
If you do want a listing file, you may want to put a .NOLIST directive before your first .INCLUDE and a .LIST

```
after your last one, so the contents of all the included
files won't appear in the listing.
```

You can assemble the code manually and then, after you've created or changed the Pascal source file, use the exec file for the application as illustrated earlier (adding the name of the assembly-language object file to the list of Linker input files). You may also want to set up an exec file that just assembles the assembly-language routines and links the resulting object file with everything else, for when you've changed only those routines and not the Pascal program. This exec file would begin with the responses listed above and then continue with step 2 of the exec file illustrated earlier.

If the entire application is written in assembly language, the source file will have the same structure as the one shown above, but at the beginning of the main program you'll have a .MAIN directive:

```
.MAIN SampA
```

Even if you have nothing to link your program with, link it by itself; the Linker will put it into a format that RMaker can accept.

SUMMARY OF PUTTING TOGETHER AN APPLICATION

This summary assumes the file-naming conventions presented in the "Getting Started" section. Page numbers indicate where details may be found.

ONE TIME ONLY:

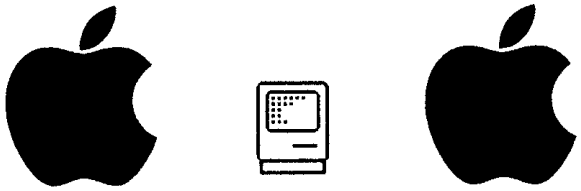
- Prepare a Macintosh system disk by copying the System Folder from the Mac Build disk to a new Macintosh disk (page 4).
- On the Lisa, use the Editor (via the Edit command) to create the exec file (page 14).

ONCE PER VERSION OF YOUR APPLICATION'S SOURCE/RESOURCES:

- On the Lisa, use the Editor to create or edit the application source file (page 6) or the Resource Compiler input file for your application's resources (page 7).
- Insert the Macintosh system disk into the Lisa.
- On the Lisa, run the exec file (page 17). It will eject the Macintosh disk when done.
- To try out your application, remove the disk from the Lisa, insert it into the Macintosh, and open the application's icon.
- When appropriate, back up your working files by using the Backup command in the File Manager to copy Work/= to Back/=, or onto a 3 1/2-inch disk (with, for example, Backup Work/= to -lower-=).

(note)

If you create an icon for your application (or modify it), you must delete the invisible desktop file (page 17).



END OF DOCUMENT